BEST AVAILABLE IMAGES

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images may include (but are not limited to):

- BLACK BORDERS
- TEXT CUT OFF AT TOP, BOTTOM OR SIDES
- FADED TEXT
- ILLEGIBLE TEXT
- SKEWED/SLANTED IMAGES
- COLORED PHOTOS
- BLACK OR VERY BLACK AND WHITE DARK PHOTOS
- GRAY SCALE DOCUMENTS

IMAGES ARE BEST AVAILABLE COPY.

As rescanning documents *will not* correct images, please do not report the images to the Image Problem Mailbox.

UNITED STATES PROVISIONAL PATENT APPLICATION

FOR

METHOD AND APPARATUS FOR ADAPTIVE WEB SERVICES

Inventors:
Garland Wong
Carlos Chue
Anthony Tang
Reza Ghanbari
Dyami Calire
Eric VanLydegraf
Meliza P. Sanchez
Trent Barnes

Prepared by:
BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN LLP
12400 Wilshire Boulevard
Seventh Floor
Los Angeles, California 90025-1026
(408) 720-8598

Attorney's Docket No: 4348P003Z

"Express Mail" mailing label number: EL627477225US
Date of Deposit: March 23 2001
I hereby certify that I am causing this paper or fee to be deposited with the United States Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Assistant Commissioner for Patents, Washington, D. C. 20231

Glenn E Von Porat
(Typed or printed name of person mailing paper or fee)

Glenn E Von Porat
(Signature of person mailing paper or fee)

March 23 2001
(Date signed)

METHOD AND APPARATUS FOR ADAPTIVE WEB SERVICES

In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the invention. It will be apparent, however, to one skilled in the art that the invention can be practiced without these specific details. In other instances, structures and devices are shown in block diagram form in order to avoid obscuring the invention.

Reference in the specification to "one embodiment" or "an embodiment" means that a particular feature, structure, or characteristic described in connection with the embodiment is included in at least one embodiment of the invention. The appearances of the phrase "in one embodiment" in various places in the specification are not necessarily all referring to the same embodiment, nor are separate or alternative embodiments mutually exclusive of other embodiments. Moreover, various features are described which may be exhibited by some embodiments and not by others. Similarly, various requirements are described which may be requirements for some embodiments but not other embodiments. The detailed examples of the following pages are of an illustrative rather than restrictive nature, and some of the requirements referred to may

be requirements for a specific example and thus not limits on the spirit and scope of the invention.

Technology Platform Developer's Guide

Kinzan, Inc.
2111 Palomar Airport Road
Carlsbad, California 92009

Release: Technology Preview 4 (TP4)

KINZAN

1.1 Introduction

In various embodiments, a goal of the Technical Preview 4 release of the Kinzan Technology Platform (KTP) is to provide web-based applications that are easily assembled and extended by developers and development partners with a wide variety of skills.

The various components of the KTP Framework are intended to untangle the web of static and active content, style, structure, application logic, templates, servlets, and persistence mechanisms that is the norm in application environments that are driven by Java Server Pages (JSP). The result represents a major commitment to flexibility, extensibility, integrability, scalability, and reliability at all levels of application development.

1.1.1 Assembling Is Easier than Building

In one embodiment, the KTP is an open technology platform that enables rapid assembly, extension, and configuration of generic application modules based on Adaptive Web Services(AWS). In one embodiment, this is achieved by modularizing style, presentation, content, application logic, business logic, and services, and providing runtime capabilities that dynamically assemble these AWS. The result is a development and deployment environment that cleanly separates the various elements required to build these Adaptive Web Services. By separating these elements, teams can work more effectively, with various team members contributing to modules that require their skill sets, rather than requiring all team members to have multiple skill sets.

The Kinzan platform enables the software engineer to work collaboratively with information architects, web developers, graphics designers, and customer advocates to create compelling network-based applications that can be delivered to various types of network devices (e.g., web phones, browsers, etc.) and with different branding and localization to different languages.

The Kinzan platform also offers a series of abstractions that modularize and generalize application development at all levels of the application architecture. The result is an environment that enables modular assembly of applications from pre-built adaptive web services, while minimizing the amount of custom work and maintenance required for different customers.

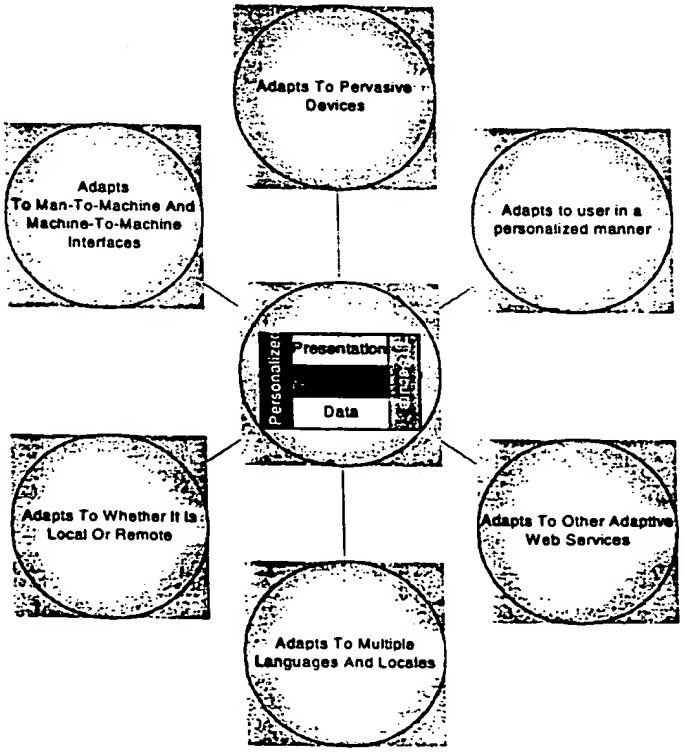
1.2 Adaptive Web Services

Kinzan believes that the next paradigm shift is to provide web services in the form of Adaptive Web Services. The KTP platform is designed and implemented with the intent of enabling developers with the framework and tools necessary to build and deploy these next generation services. We will move from a world of monolithic tightly coupled applications to a world of distributed loosely coupled applications comprised and built with Adaptive Web Services.

When object oriented languages such as C++ emerged, more rapid application development could be realized because of code reuse and applications that are more reliable could be built because they could be tested at an object level. The limitation is that this technology requires very technical individuals to develop these applications.

1.2.1 Elements of Adaptive Web Services.

The following diagram outlines the elements of an embodiment of an Adaptive Web Service.



Kinzan's Adaptive Web Services are comprised of five main elements: presentation, code, data, personalization attributes, and security. Each element of Adaptive Web Services will provide programmers and business people the ability to control the functionality of a service being provided to users.

1.2.2 Adaptive Web Services Adaptability

Kinzan's Adaptive Web Services are intelligent software components that can conform to how they are accessed and the context that the adaptive web services provide. Adaptability may exist in the following areas:

- Adapts to devices accessing the service such as: TV's, PDA's, Cell Phones, etc.
- Adapts to other Web Services that surround it and integrates via messaging
- Adapts to language and locale of the browser
- Adapts to location where it is loaded whether remote or local
- Adapts to man-to man or man-to machine interface
- Adapts to user (personalization).

1.3 Motivation

Fundamentally, the KTP framework is intended to enable rapid branding, assembly, extension, and configuration of generic application modules. It does so by modularizing style, presentation, content, application logic, business logic, and services, and providing runtime capabilities that dynamically assemble these modules into rich applications.

1.3.1 Market Drivers

Current server-based application development technologies tightly couple presentation, content, and logic, making the modular development of brandable applications impractical. Each application becomes a one-time implementation, compounding support and maintenance issues going forward.

Kinzan has a long history of delivering uniquely branded and configured Internet content and commerce-management systems to our customers on our hosted platform. The result is a deep understanding of application frameworks that enable the rapid branding and configuring of generic applications.

Competitive pressures mean time to delivery is key. This places a premium on being able to rapidly and easily assemble, reconfigure, and extend generic applications. The global nature of the Internet also places a premium on easy localization and support for emerging alternative devices for Internet access (browsers, cell phones, PDA's, pagers, etc.).

Finally, supporting an application development and deployment environment in a shared hosting facility requires robust security and reliability to support multiple applications and development partners simultaneously.

1.3.2 Technology Drivers

Even the most sophisticated technologies have limited value if they are too complex to use. There is a critical need to develop and package these technologies in a way that makes them accessible to teams with varying technical skills.

Typical client-server implementations separate the client from the database. In the JSP world, that means direct connectivity between the JSP and the database via Java Database Connectivity (JDBC) queries (also known as Java Model 1 architecture).

With Enterprise Java Beans (EJB's), developers may abstract out business logic into a separate services layer using entity EJB's and session EJB's. However, that still leaves presentation and application logic in the JSP layer. In JSP 1.1,

using tag libraries helps to extract the application and presentation logic associated with common components out of the JSP; but the different elements are still co-mingled. Combining JSP's and servlets in a Model-View-Controller (MVC) configuration (also known as Java Model 2 architecture) alleviates some issues, but generally requires a higher level of complexity for all developers on a project.

The KTP framework (including rendering, state management, and services management processes) cleanly separates style, layout, presentation, and content from application and business logic. It also supports localization and output to target deployment platforms with different capabilities (browsers, cell phones, PDA's, pagers, etc.).

The collection of files that make up a Kinzan Page Descriptor (KPD) is the "source code" that is "compiled" into a series of modular servlets (and associated data), which are then dynamically assembled by the rendering engine at runtime. In this way, developers can take full advantage of the power and performance of Java servlets, while benefiting from increased reuse and modularity and the clean separation of function and responsibilities in the development process.

The Kinzan State Manager uses XML-based Kinzan State Diagrams (KSD) to connect and configure application logic modules, effectively assembling the controller tier of the application.

The Kinzan Services Manager provides an abstraction layer for many kinds of back-end services and data and includes the ability to dynamically look up and bind to services. The result is modularization of the model tier.

The KTP Framework is flexible. If developers choose to implement their pages "close to the metal" as standard JSP pages or Java servlets, they may. However, to do so is to give up the benefits of being able to apply different style and branding elements to a given application. In practice, developers will likely leverage the KTP Framework for elements of their applications that require flexible styling and branding and apply conventional HTML, XML, and JSP techniques for sections where the flexibility is not required.

The KTP framework is designed to support the development and deployment of closely related applications across multiple capability devices (browsers, cell phones, PDA's, pagers, etc.).

As a practical matter, it is unlikely that there will ever be a painless way to simultaneously deploy a single dynamic web application to standard web browsers and a two-way pager with a 20-character display, no matter how sophisticated the development system. However, the KTP framework allows developers to rapidly develop and deploy closely related applications to platforms with different capabilities. By only needing to change the presentation modules, developers can leverage a common development methodology, business logic, localization, and data persistence layers among all applications.

1.4 Kinzan Technology Platform Overview

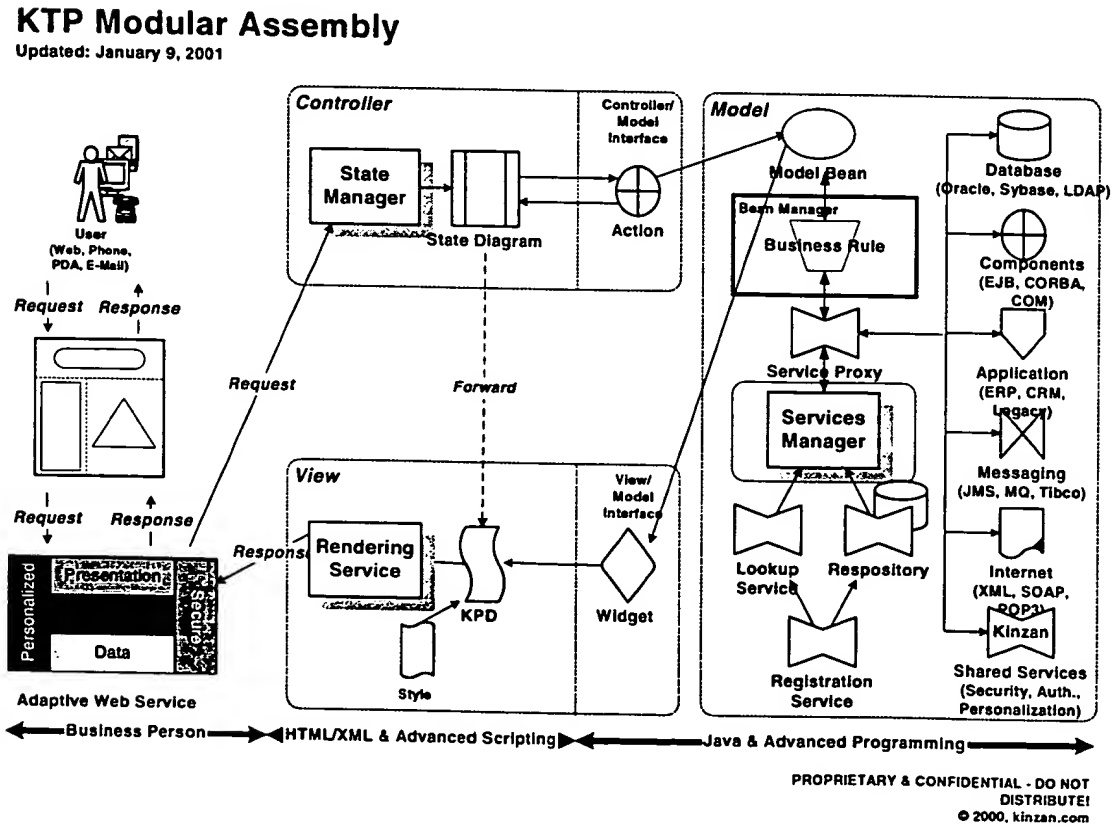
At the base is the Kinzan Infrastructure Platform. This represents the high-availability, high-scalability hosting infrastructure for the KTP. These data centers can be shared among many partners and customers, giving each access to more capacity and more reliability at less cost.

The Kinzan Software Platform represents the core services and capabilities described in this document. It is a shared foundation for all applications built on top of the KTP.

This core platform can also be extended by integrating in new services, then exposing these services with widgets and KSD's.

The modular nature of the KTP encourages reuse at all levels. Collections of modules (widgets, KSD's, and support modules) form libraries that may be used to streamline application development.

1.4.1 Modular Assembly Approach



KTP Modular Assembly

The KTP framework offers a modular assembly approach to page and application definition. By leveraging real-time modular assembly of pages by the Rendering System, and modular assembly and control of applications by the Kinzan State Manager, KTP-based applications are intrinsically extremely flexible and configurable.

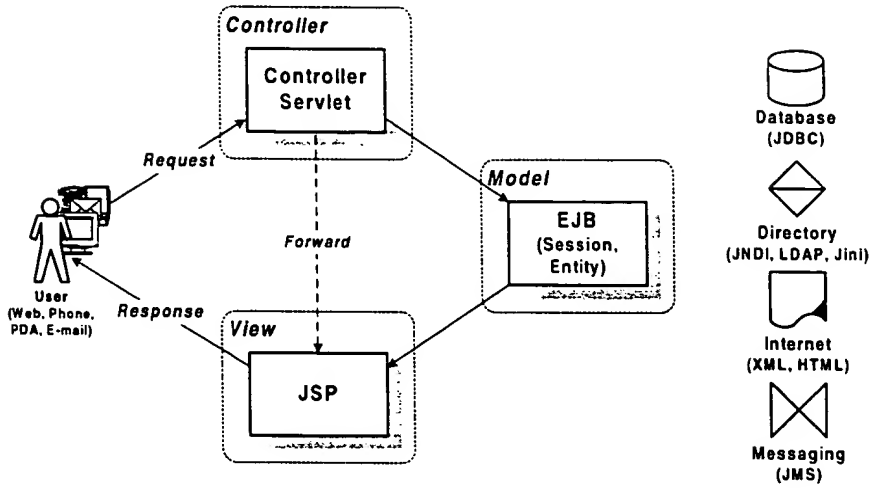
The modular assembly approach also empowers junior team members to assemble sophisticated applications, rather than having to code the various elements themselves. More importantly, it makes it practical to develop tools and applications that directly manipulate and configure other applications, empowering customers to maintain their own applications via "codeless" development.

The KTP framework decomposes style, structure, presentation, active content, branded assets, application logic, business logic, and various support services into distinct modules. Development team members contribute by developing modules that are appropriate to their skill set. These modules are then assembled to implement web pages and web applications.

The Kinzan Page Descriptor (KPD) is the artifact that describes the modules to be used to assemble a page, and the Kinzan State Diagram (KSD) is the artifact that describes how pages and application logic are to be assembled into dynamic web applications.

Although the KTP framework attempts to isolate dependencies between the various element types, there are, however, situations where rules are necessary to control the coupling between element types. Examples are included in the next sections. In these cases, the KTP framework allows definition of module variants that are bound to any combination of style, locale, and/or target device. Changing any of these instantly changes the module variant that is used when rendering a page or driving the application.

1.4.2 Request-Response Architecture



Request-Response: Java Model 2 Architecture

In the Java Model 2 architecture, Java Server Pages (JSP's) are used to query Enterprise Java Beans (EJB's) to generate a dynamic web page for a user. Different JSP's may be written to output different XML variants to support different kinds of devices.

HTML is embedded in the JSP, along with the Java programming that is required to access the EJB's. Since sophisticated developers and sophisticated web designers are rarely the same person, it is often difficult to support a collaborative development environment.

When a user performs an action (say, clicking the check out button at an e-commerce site), the web page passes the action to a Java controller servlet. This servlet processes the action, stores information in EJB's if necessary, and then requests that the next appropriate JSP to be rendered.

The KTP framework takes each of these layers and modularizes it. At the view layer, the Rendering Service dynamically assembles pages from all the modules that make up the page. Interfacing with the model layer is managed separately from the view layer, minimizing the impact on those that are more expert with graphics design and page presentation.

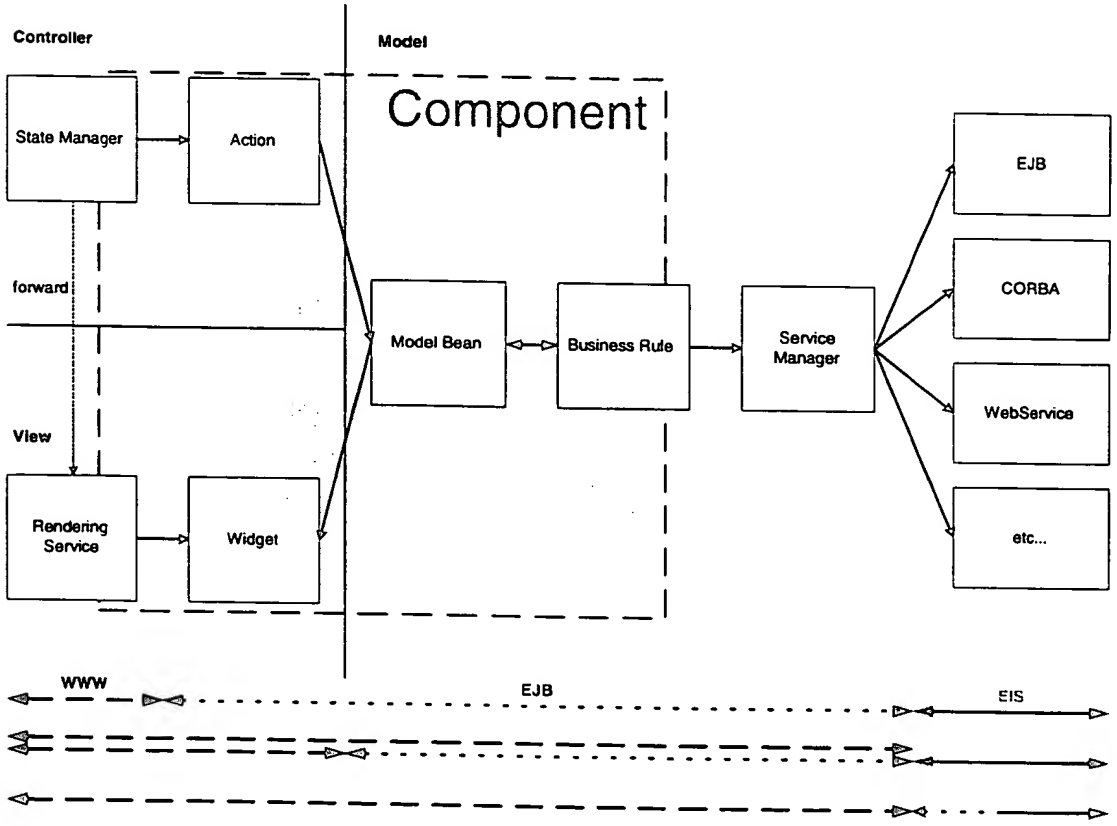
The State Manager performs a similar function at the controller layer. Kinzan State Diagrams are used to dynamically assemble and configure application-logic modules called action components. Interfacing with the model layer is managed with action components, allowing information architects and site designers to more easily manage the configuration of application flow.

The Services Manager provides an abstraction and resolution layer on top of many kinds of services and components, making it easier for components and application-logic developers to take advantage of rich back-end services at the model layer with minimal programming.

1.5 The Component Overview

Up to this point, we have spent a considerable amount of time explaining Kinzan's vision and how that vision solves problems for Kinzan's developers and partners. What follows is an explanation of the Component Architecture.

1.5.1 KTP Component Overview



KTP Component Overview

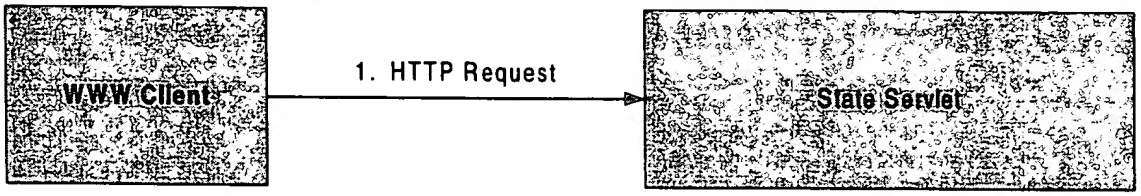
The KTP Component architecture, as represented by the above figure, is structured so that the component loosely couples the 3 layers of the Model 2 architecture. The component is comprised of a logical grouping of actions (Controller Layer), widgets (View Layer), and model beans (Model Layer). The sum of these three objects when executed by the State Manager, Rendering Service, and Model Manager make it possible for seemingly dissimilar objects to behave as one object to perform an action or some set of actions.

Additionally, the KTP architecture can be deployed in several different configurations thereby affecting the behavior and functionality of the components. One such configuration is to have the state manager and rendering service executing on the first tier (web server) with the rest of the objects executing on the middle tier (application server). Another possible configuration is to execute the state manager and rendering service on the first tier and model bean and service manager on the first tier or middle tier. This is made possible by Kinzan's innovative approach of integrating an internal EJB container into the KTP platform. Finally, armed with the knowledge of the EJB container being integrated into the KTP platform, we can deploy and execute the whole platform in the first tier. This gives the KTP platform added flexibility and scalability for the small yet diverse production facility and for the developers who are leveraging the KTP framework.

We will discuss interactions among the State Manager (controller), Rendering Service (view), and Model Manager (model) in this section; subsequent sections will describe the specific features of each layer.

1.5.2 Component Request

NOTE: Figures in this section are sequenced such that the message numbering depends on previous figures.



KTP HTTP Request

The HTTP Request is in the form:

```
http(s)://address:port/stateServletName/application/ksd/state?componentID=CID&EVENT=event
```

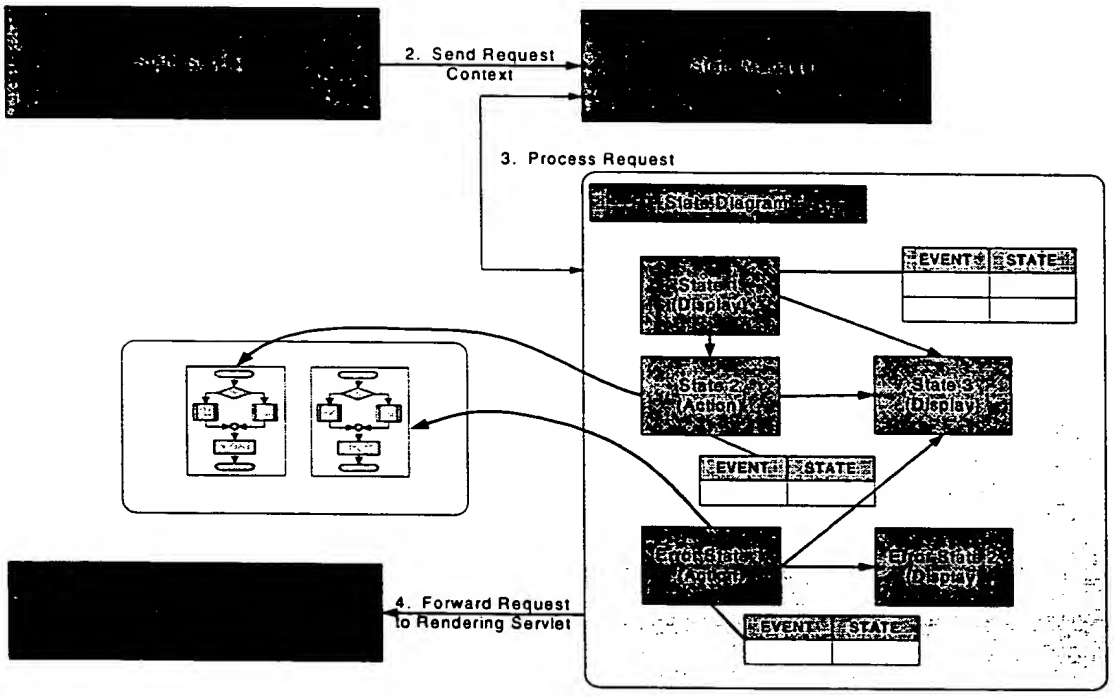
where:

```
stateServletName = alias for state servlet in the servlet runner
application = name of the application/site in which the state diagram exists
ksd = name of the state diagram to run
state = state in the state diagram to run
CID = ID for component to take action on
event = name of the event to post
```

Typically, an application that is in process will provide the CID, which automatically identifies the application, KSD, and event name to the state manager. The KSD also contains a default state so that state does not need to

be specified. The CID and event name are the keys, which allow the State Manager to identify the next state to execute or display.

1.5.3 Request Processing



State Manager Processing HTTP Request

1.5.4 State Diagram Detail

Furthermore, the user context is sent to the State Manager, which queries it to determine the appropriate actions to be performed by the state manager on behalf of the user. When all actions have been completed, the State Manager forwards the results to the rendering engine (servlet) which forwards it to whatever device the user sent the request.

Example State Diagram:

```
<ktp:stateDiagram name="Login" startState="Login" directAccess="false" visibility="public">

    <ktp:displayState name="State1" kpd="KPD1" directAccess="true" >
        <ktp:transition event="Event1"  state="State2" />
        <ktp:transition event="Event2" state="State3" />
    </ktp:displayState>

    <ktp:action name="State2" component="net.kinzan.component.Component1">
        <ktp:transition event="Event3" state="State3" />
    </ktp:action>
```


KINZAN

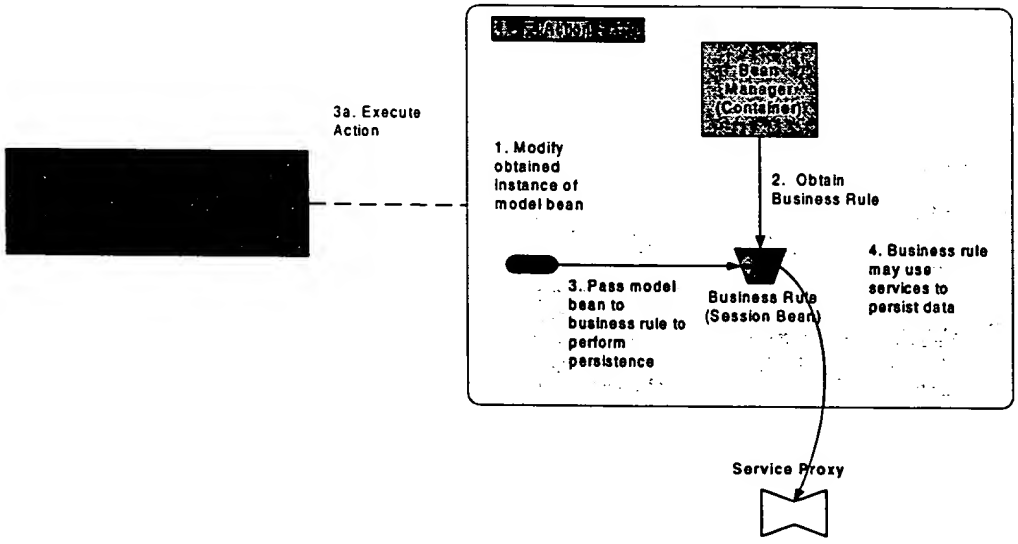
```
    <ktp:displayState name="State3" kpd="KPD2" />

    <ktp:action name="failure" component="net.kinzan.component.Component1">
        <ktp:transition event="Event4"  state="State3" />
        <ktp:transition event="Event5" state="failure2" />
    </ktp:action>

    <ktp:displayState name="failure2" kpd="defaults/failure"/>

</ktp:stateDiagram>
```

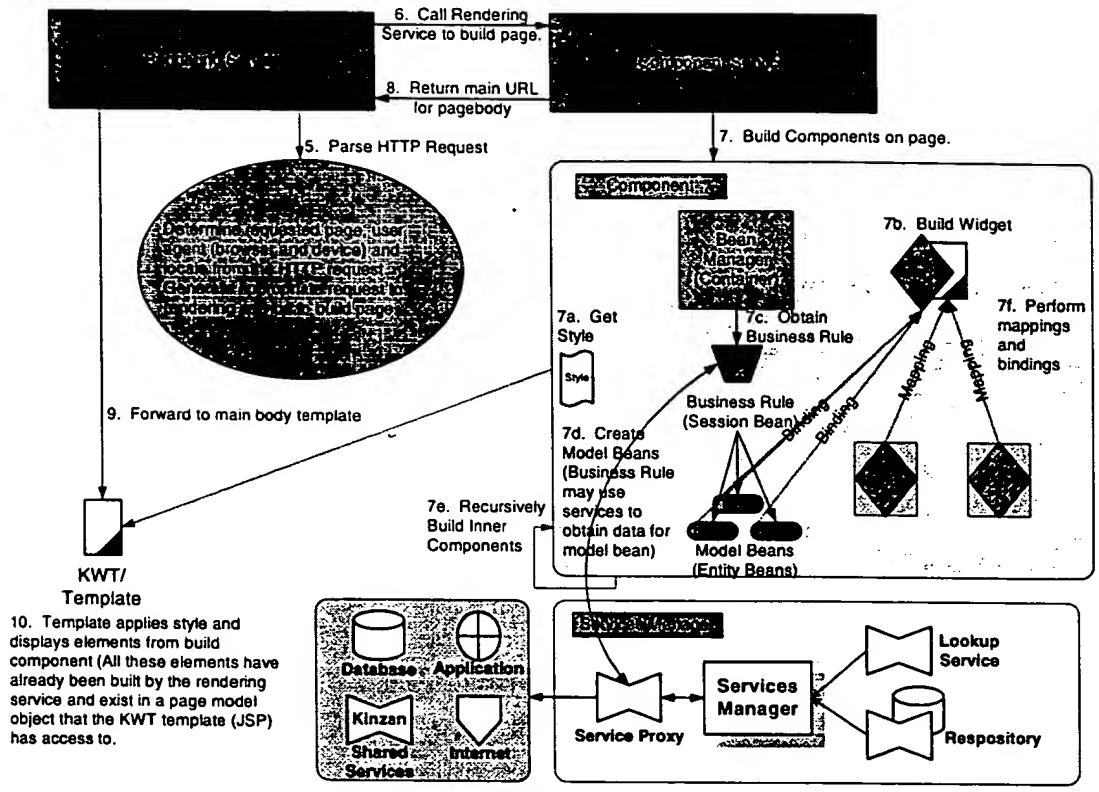
1.5.5 Component Performing an Action



KTP Component Performing an Action

In the KTP an action is performed on behalf of the user when the State Manager executes a specific action. However, the State Manager may need to make changes to the model to perform the appropriate action on behalf of the user. For example, an action might need to update the stock symbols list for a given account. If this is the case, the action, through the component, should be able to obtain the appropriate business rules, which supply the appropriate model beans to make the required changes as depicted in the above figure.

1.5.6 Rendering the Compon nt



KTP Rendering the Component

When the rendering servlet receives a request, it will determine the page requested from the URL of the request (the client will never directly access the rendering servlet), which is in the following form:

```
http://address:port/renderServletName/application/kpd
```

where:

```
application = the name of the application from which this page comes
kpd = the name of the page (Kinzan Page Descriptor)
```

With this information, the component service is able to look up the required page in the database, find the root component of the page, and build the page. This is done by building each component on the page from the bottom up, allowing for bindings from parent components to child components to be resolved correctly. For each component built, the component service will perform the required bindings, calling the appropriate model beans for model data as necessary, and building the component tree so that the parent-child relationships are correct for

KINZAN

the current mode of each component. After the complete page is built, control is forwarded to the main body template of the page. This template has the variants about the device that is making the request and includes the nested templates that render the components on the page. This is done by obtaining the current widget for each component (set into the component by the component service based on mode), and getting the appropriate template for that widget by device type, locale, and style in the same fashion. Each template may also include other assets, such as images, obtained from the widget. Style is applied when the template is rendered by the JSP engine of the web server, as they are encoded to expand the appropriate tag attributes for the current style where required.

The Widget

Example of a widget:

```
<ktp:widget name="sampleWidget">
    <!-- The attribute list defines bindable attributes in the widget. A class
         is automatically generated from the attribute list. -->
        <ktp:attributeList>
            <ktp:local name="text" type="java.util.String"/>
        </ktp:attributeList>

        <!-- style, device, and locale attributes default true and indicate
             to the preprocessor whether such variants should be searched for -->
        <ktp:widgetTemplate>sampleTemplate.kwt</widgetTemplate>
</ktp:widget>
```

The Widget Template

Example of a Widget Template:

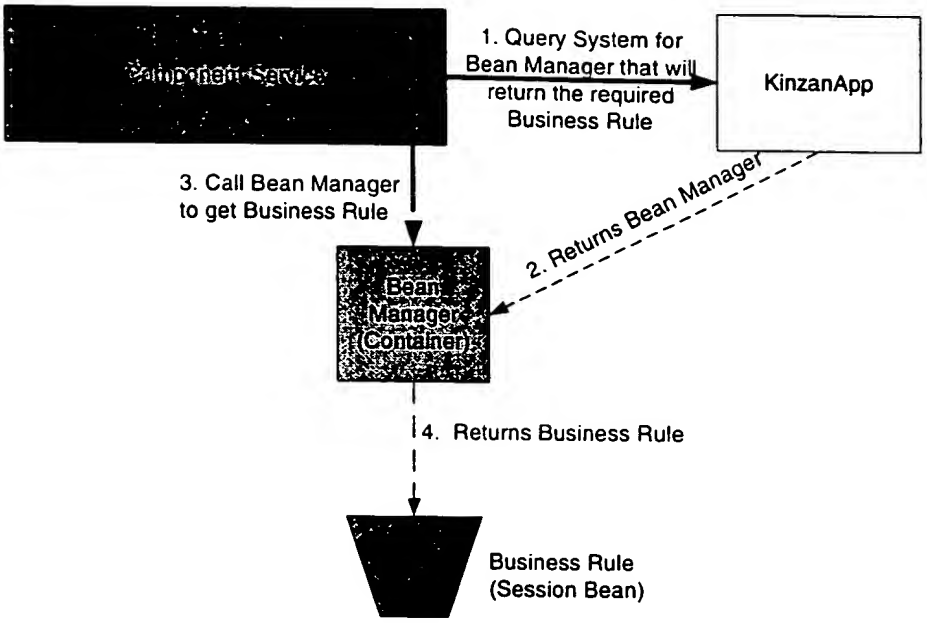
```
<!- Sample KWT file for simple widget example -->
<table>
    <tr><td><ktp:zone name="zone1"/></td></tr>
    <tr><td><ktp:text text="text"/></td></tr>
</table>
```

where:

- **Ktp:Zone** is a placeholder for other components to be **mapped** into
- **Ktp:Text** places the value of the named attribute from the widget into the template with the appropriate styling for the current style.

The Kinzan Widget Template (KWT) can be as simple as an HTML fragment or as complex as a JSP file. The KTP provides several tags which allow a template to access widget attributes, define appropriate actions to post to the controller, and defines zones to embed nested components. The loader processes the KWT into a JSP containing the appropriate JSP code to perform the required mappings. We will discuss mappings after first explaining Bean Managers and Business Rules.

1.5.7 The Bean Manager



Bean Manager Interaction

The rendering service, in performing bindings, may need data from the model as detailed in the above figure. This is obtained from model beans, which are returned by business rules, which are returned by bean managers. A bean manager is responsible for registering with the system the business rules it manages. These business rules, which are stateless session beans, are referenced in the KApp file in the model bean definition, along with the method to call and required parameters to obtain the requested model bean. The rendering service has this information available and is able to query the system to obtain the appropriate bean manager from which it can obtain the business rule required to generate the model bean. The business rules typically contains the business logic, which facilitates the formatting, filtering, or massaging of the data returned from the model in a useful format to the widget requiring it. Additionally, the rendering service may perform transformations to transform a given model bean or model bean attribute into the type required for binding to the widget attribute successfully.

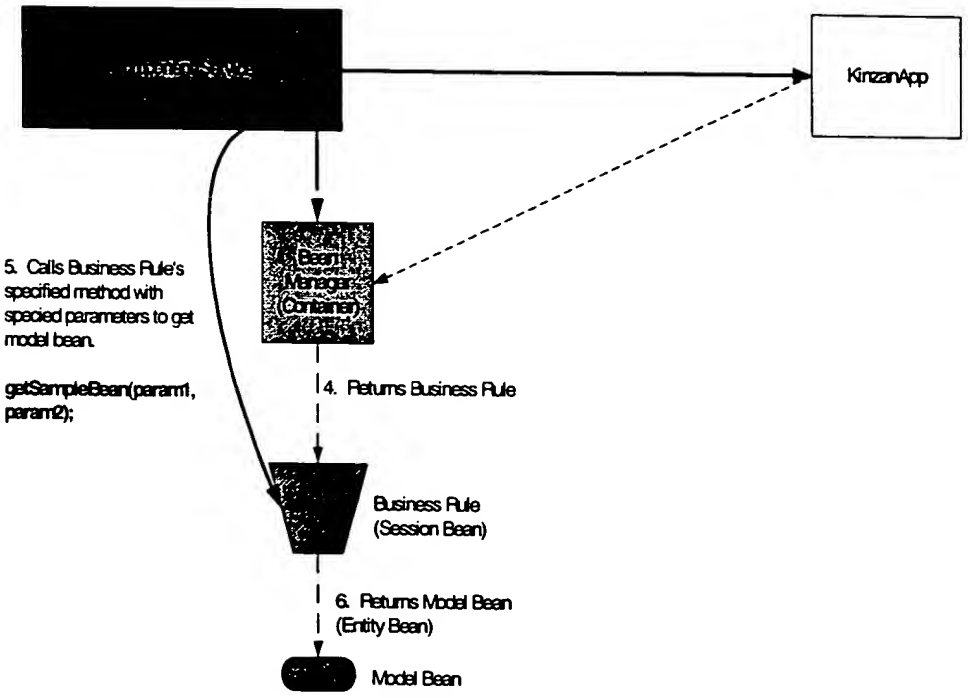
The following is an example of using a Bean Manager in the KApp file:

```
<ktp:beanManager name="SampleBeanManager">
```

The Bean Manager is referenced by name, for example: "SampleBeanManager".

Note:
Bean Managers run as services in the model tier.

1.5.8 The Business Rule (1)



Business Rule Interaction

Example of a business rule class referenced in a KApp file:

```
<ktp:modelBean name="sampleBean">
    <ktp:rule name="sampleBusinessRule" method="getSampleBean"/>
    <ktp:param name="param1" type="java.lang.String">
        <ktp:from widget="sampleWidget" attribute="text"/>
    </ktp:param>
    <ktp:param name="param2" type="java.lang.String">
        <ktp:from literal="param2"/>
    </ktp:param>
</ktp:modelBean>
```

The model bean declaration also declares the business rule that will return the required model data as well as the method in the business rule to call, and the parameters to pass. At runtime, when the system is building the component, the business rule's method is called, and the parameters are passed to obtain the model bean data required.

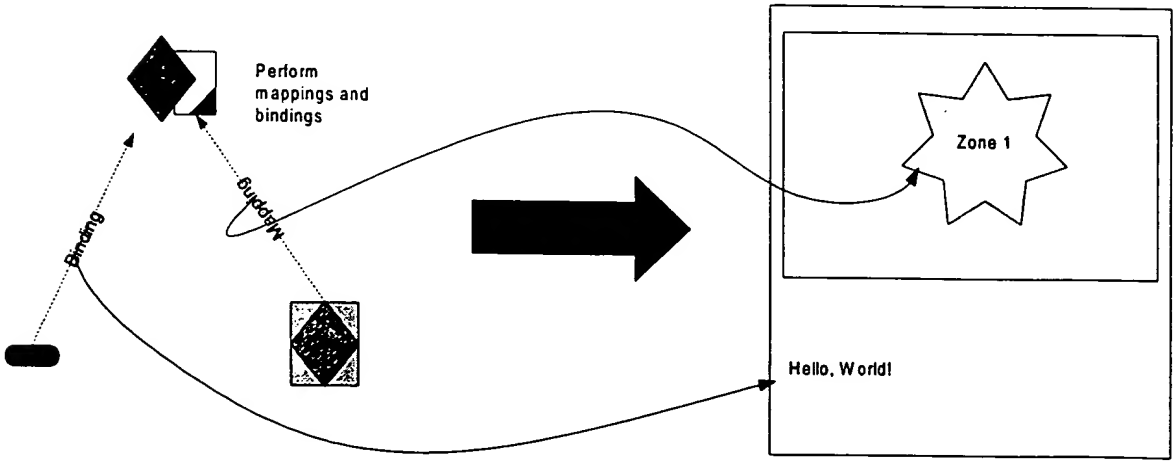
1.5.9 The Business Rule(2)

The following is an example of a Business rule class:

```
public class SampleBusinessRule extends KinzanBusinessRule
{
    ...
    public SampleBean getSampleBean( java.lang.String param1, java.lang.String param2 )
    {
        InitialContext ic = new InitialContext();
         // Get home interface
        SampleBeanHome sbh = (SampleBeanHome)ic.lookup("SampleBeanHome");
        // Get remote interface to entity bean.  param1, param2 form the primary key.
        SampleBeanPK pk = new SampleBeanPK();
        pk.key1 = param1;
        pk.key2 = param2;
        SampleBean sb = sbh.findByPrimaryKey( pk );
        return sb;
    }
    ...
}
```

NOTE: That a business rule may return a simple Java Object, or a more complicated entity, such as a remote interface to an entity bean.

1.5.10 Binding and Mappings



Component Binding and Mapping

The KTP component service, upon determining the current mode for a given component, needs to determine if in that mode, the widget has zones, which require nested components. If this is the case, then the component service is

responsible for mapping the child components and building them, so that when the widget template includes them, they will display correctly.

Example of a Component binding and mapping from a KApp file:

```
...
<ktp:componentModeList>
    <ktp:componentMode widget="sampleWidget">
        <ktp:zone name="zone1" component="anotherComponent"/>
    </ktp:componentMode>
</ktp:componentModeList>
```

The component service is also responsible for binding data from the model tier to the widget attributes for display.

1.5.11 Component Security

Component Mode Security (Level 2)

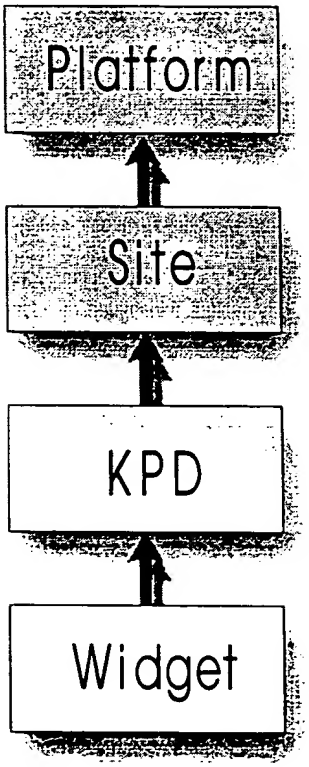
A component builder can set up permissions on the modes of the component compile time. The application builder will be responsible for mapping application security roles to component permissions (Refer to Appendix A).

1.6 The Kinzan Application

1.6.1 The Kinzan Application Hierarchy

Within the Kinzan Technology Platform, sites are arranged hierarchically. Resources may be inherited or overridden at each level.

Each Kinzan Web site is actually part of a large hierarchy of sites. Resources associated with sites at upper levels are available to sites at all lower levels. Lower levels may provide their own variant for a resource, overriding the more global declaration. This hierarchical arrangement allows you to build libraries of resources, streamlining development and facilitating the reuse of common modules.



Generic resources visible to all applications and pages developed on the platform.

Changes here are automatically and immediately applied throughout the hierarchy unless the resource has been overridden at a lower level.

Resources associated with a particular site or application—template bundles that represent all the able to be branded resources in an application. Sites may also be bound to parent sites; all resources available to parent sites are recursively available to child sites.

Resources required for rendering a particular page, including other KPDs (if necessary). Changes here occur only on the specific page

Resources specific to a particular widget instance. This is the termination point of the KTP hierarchy.

Resources at a lower level override identically named resources declared at a higher level. For example, if the platform provides a generic widget named *stockQuote*, and you want to replace it with a different implementation, you need only declare a new widget named *stockQuote* at a lower level in the hierarchy. You may decide to implement a different widget, one named *myStockQuote* for

example, using the new widget requires you to modify all references to *stockQuote* within the various KPDs in the application.

This is not an issue when you are developing an entirely new application. However, it is more common to begin with a generic implementation of the application and customize it for a particular client. In this case, overriding the resource at the application level applies the customization to the entire application at once.

At the same time, a resource may be marked such that overriding it is not allowed. This is useful for limiting the behavior of children sites.

1.6.2 The KApp File and Loader

An application, or site, is described by the Kinzan Application file (KApp file). This file contains definitions for the various resources required by the application, and may include other KApp files to import their resources as well. Defined within the KApp file are styles, widgets, assets, components, and pages (KPD's). In addition, for an application, the domain name and document root that identifies the site are defined. Sample KApp files can be found in Appendix A.

While the KApp file defines resources used at runtime, the runtime environment does not deal with the KApp file directly. Instead, the KApp file is read by a loader tool, the KApp Loader, which parses the resource definitions and loads them into runtime. The loader, run at the application compile-time, is also responsible for parsing KWT files into JSP files and resolving variants for assets. Whenever the KApp file is changed, the loader must be run to make those changes available to the runtime environment. This pre-processing allows the runtime environment to run more efficiently while maintaining its dynamic nature, and gives the application developer application resource definitions in a file format that is easier to modify and maintain than the database.

At the same time, applications are fully dynamic in the runtime environment and can be manipulated and changed by appropriate tools without running the loader.

1.6.3 Organizing Files and Folders

The following illustration outlines the development directory tree structure convention used for the Kinzan platform.

domain
parent site
child site
/childSiteDocRoot
childSite.kapp
login.ksd — Kinzan State Diagram files.
/asset — Images, JSPs. HTML, and

- /image
- /de
  - /image
- /es
  - /image
- /ja
  - /image
- /fr
  - /image
- /spring
  - /image
- /fall
  - /image

other asset variants that require dynamic resolution at run time based on locale, style, or target device.

- /kpd
  - home.kpd
  - login.kpd
  - contactUs.kpd

Kinzan Page Descriptor files.

- /kwt
  - /companyDefault
  - /spring
  - /fall

Kinzan Widget Template files.

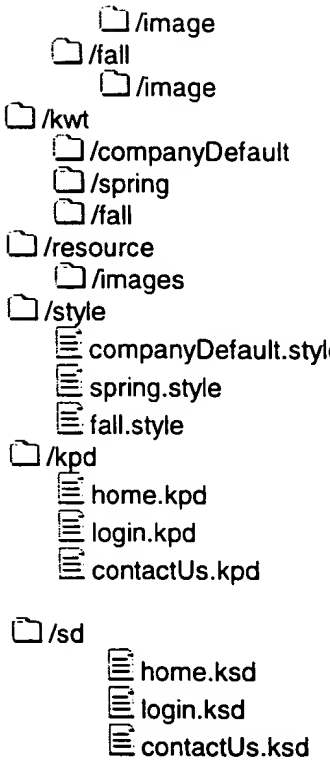
- /style
  - companyDefault.style
  - spring.style
  - fall.style

Server-side style files that are similar to cascading style sheets.

The following illustration outlines the deployment directory tree structure convention used on the Kinzan platform

- webroot
  - parent site
    - child site
    - /childSiteDocRoot
      - childSite.kapp
      - /asset
        - /image
        - /de
          - /image
        - /es
          - /image
        - /ja
          - /image
        - /fr
          - /image
        - /spring

The loader resolves these variants at compile time. So at runtime the file system is not searched. The page model can be queried to retrieve the correct variant's URI.



At compile-time and at runtime, the search path for the site or application includes the search root for the site or application and for each of its parent sites. The search root defines where files for the site are located. The search order is as follows:

1. Look for the file in the *root*/`[subdir]` directory. The type of file being searched for determines which subdirectory is searched. See the tree diagram (above) for specific information about the expected directory structure.

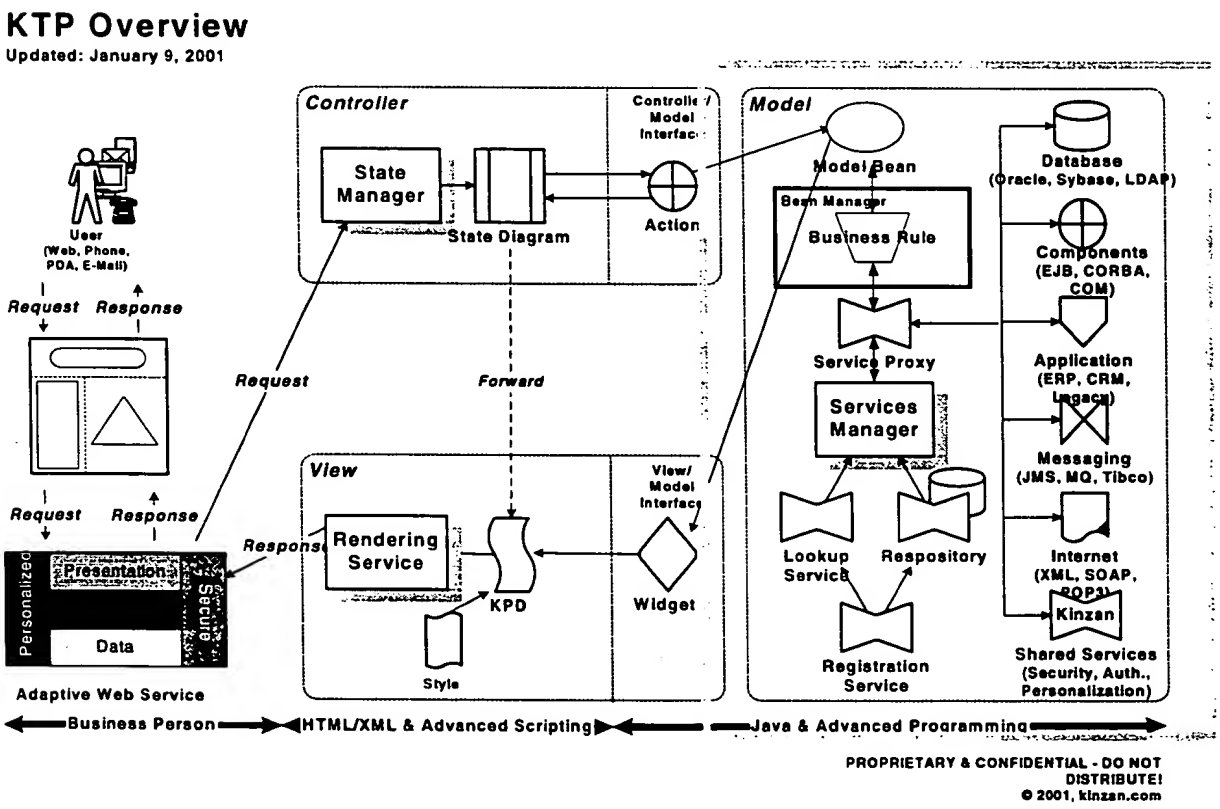
2. Search through the library search path for the KApp Loader in the same fashion.

For resources that support locale, style, and/or target device variants, their variants need to be placed in subdirectories within the appropriate `[subdir]` using the pattern `[locale]/[device]/[style]`. The loader will search through each of these directories to obtain the relevant variants for the resource.

1.7 The Model Layer

The classic definition of Model/View/Controller architecture describes the model layer as the internal workings of the software. Typically, this layer includes messaging, data storage, integration with legacy systems, and so on.

In the KTP architecture, the model layer consists of the Services Manager and myriad services that are integrated into the Kinzan platform. Examples of integrated services include database systems; EJB, CORBA, and COM components; and external applications such as credit card processing systems and commercial package-tracking systems.



1.7.1 Overview

When you work within the model layer, you will typically develop an application to integrate into the KTP or develop the glue layer that will link an external application or service into the KTP.

KINZAN

Skill Level:

Developing new services and integrating external services into the Kinzan platform requires intermediate to advanced Java programming skills.

1.7.2 Developing Services

When you develop a service, you will both develop an application and integrate it into the KTP or you will develop the glue layer that will link an already developed application or service into the KTP.

Services may be local or remote.

- Local services are collocated—executed in the same memory space as the application that includes them. Local services can also be considered proxyless. Proxyless services do not need to communicate with a back end to perform their functions; all the functionality is contained in the downloaded JAR.
- Remote services are not collocated; invoking their operations requires remote calls. Remote services may be either proxy services or SOAP-enabled services.
    - Proxy services rely on the functionality residing in the back end to perform their functions. The classes in the downloaded JAR communicate with and invoke operations on a remote server.
    - SOAP-enabled services are supported within the native functionality of the platform. They do not require any JAR's to be downloaded because the platform automatically generates all classes that are required to communicate with the back end using SOAP.

To integrate a service into the KTP, you must implement the `KinzanService` interface. This interface declares methods that initialize and close down a service as well as set and get the service name and the service manager for the service.

The `GenericService` abstract base class implements all of the methods declared in the `KinzanService` interface except for `init()` and `fini()`. We recommend that you extend the `GenericService` class when you develop a new service rather than implementing the `KinzanService` interface directly.

Note that accessor and mutator methods within the `GenericService` class are assigned a visibility of `final`. `setName()` and `setServiceManager()` are callback methods that set the name of the service within the framework before the `init()` method is called.

When you extend the `GenericService` class, implement the `init()` method, entering any one-time operations that must be performed before a user can access the service. For remote services, bootstrap the communications protocol in `init()`.

Implement the `fini()` method to close down the service, terminate remote connections, and perform other one-time cleanup operations.

Extend the class to include application functionality, perform calls to an external API, or call those functions in another class. Additional implementation details are dependent on the type of service or application being integrated.

Creating a SOAP-Enabled Service

In general, follow the procedures below to create a SOAP-enabled service:

- Define an interface for the service. This interface should extend the `KinzanService` and the `SoapService` interfaces. (Note that `SoapService` is a marker interface, so no implementation is necessary. When the platform resolves a service that implements the `SoapService` interface, it automatically creates a runtime proxy for the service that implements the SOAP-specific calls.)
- If your interface passes or returns Java objects, these objects should implement the `Serializable` interface.
- Implement the class for your interface. The class should extend the `GenericService` class and implement the interface you defined above.
- Handle all state changes and object communications through the interface. The generic limitations of all RPC mechanisms make this necessary. For example, it would be difficult to SOAP-enable the connection pool service because the connection pool makes direct calls to its connections.

1.7.3 Deploying Services

As they relate to the platform, there are two types of services: static and dynamic.

- Static services are declared in the configuration file for the application, so are known to the application at startup. Static services may or may not be loaded into memory at startup.
- Dynamic services are discovered at runtime. Dynamic services allow you to add functionality without shutting down the server. There are three types of dynamic services.

Deploying Static Services

Static services are declared in the XML configuration file for the application. Once the service is declared it becomes available for use at startup. It may be loaded into memory immediately (loadAtStartUp="yes") or when it is needed (loadAtStartUp="no"). The following example declares the logging service and loads it into memory.

```
<service id="LoggingService" loadAtStartUp="yes">
  <class>net.kinzan.logging.LoggingService</class>
  <attribute name="propertiesFile"
                        value="@fileURLPrefix@@wariniroot@/LoggingService.properties" />
  </service>
```

Deploying Dynamic Services

Deploying dynamic services is a two-step process.

1. Register the service with one or more lookup servers.

2. Deploy the service's files into the core repository.

Registering the Service

If an application needs to call a service that is not in its configuration file, it consults one or more lookup servers until it locates the appropriate service. Registering a service with a lookup server allows you to add functionality to an application without needing to stop and restart it.

Leasing a Service

Once a service is registered with a lookup server, it remains registered until you explicitly remove it from the service or until the server is rebooted. Alternatively, you may lease a service to the lookup server. A leased service is available for a specified period of time, after which it is automatically removed from the lookup server.

NOTE:
As of this writing, the client site may still be able to use the service proxy after the service has expired. This will change when remote events are fully implemented.

Using Multiple Codebases

Using multiple codebases, it is not necessary to package the service proxy and its dependent classes in a single jar. Codebases function in a way that is similar to the CLASSPATH environment variable. Each codebase entry specifies a server and a jar. When the service manager resolves the class dependencies for the service proxy, it looks first at its CLASSPATH; if the dependency is not found, it looks at each codebase in the order specified in the registration file.

Using the Registration Tool

The registration tool is a command line utility that accepts an XML file as input. Use this tool to register and remove a service from one or more lookup servers.

Before you can use the registration tool, you must create an XML file to use as input. Following is a sample input file:

```
<bootStrap>
    <group name="TestInstance">

        <lookupService>http://server_address:port/soap/rpcrouter</lookupService>

        <service id="StockQuoteService" lease="1186400">
            <codeBase>http://server_address:port/codebase/stockquote.jar</codeBase>
            <codeBase>http://server_address:port/codebase/weblogic51.jar</codeBase>
            <codeBase>http://server_address:port/codebase/weblogicaux.jar</codeBase>

<interface>com.kinzan.example.stockquote.service.StockquoteService</interface>
            <class>com.kinzan.example.stockquote.service.RemoteStockQuoteService</class>
            <attribute name="propertiesFile"
value="http://server_address:port/codebase/StockQuoteService.properties"/>
        </service>

    </group>
</bootStrap>
```

group Organizes services into groups. Each KinzanApp is associated with only one group.

lookupSevice Specifies the location of the Lookup service. You can specify as many lookup services as you want; the registration tool contacts each specified lookup service and registers each service in the group.

service Defines the service. The service id attribute must be unique within the group. The lease attribute is optional. Specify the lease value in seconds.

codeBase Specifies the location and name of the service proxy jar or its dependent classes. Codebase entries are not required for SOAP-enabled services.

interface Specifies the interfaces implemented by this service. Be sure that each of the specified interfaces is implemented; the tool does not perform any checking.

class Specifies the class name that implements the service proxy.

attribute Defines one or more attributes for this service.

After creating the input file described above, use a command similar to the following to register the service:

```
java com.kinzan.app.service.registration.Tool -R inputFile.xml
```

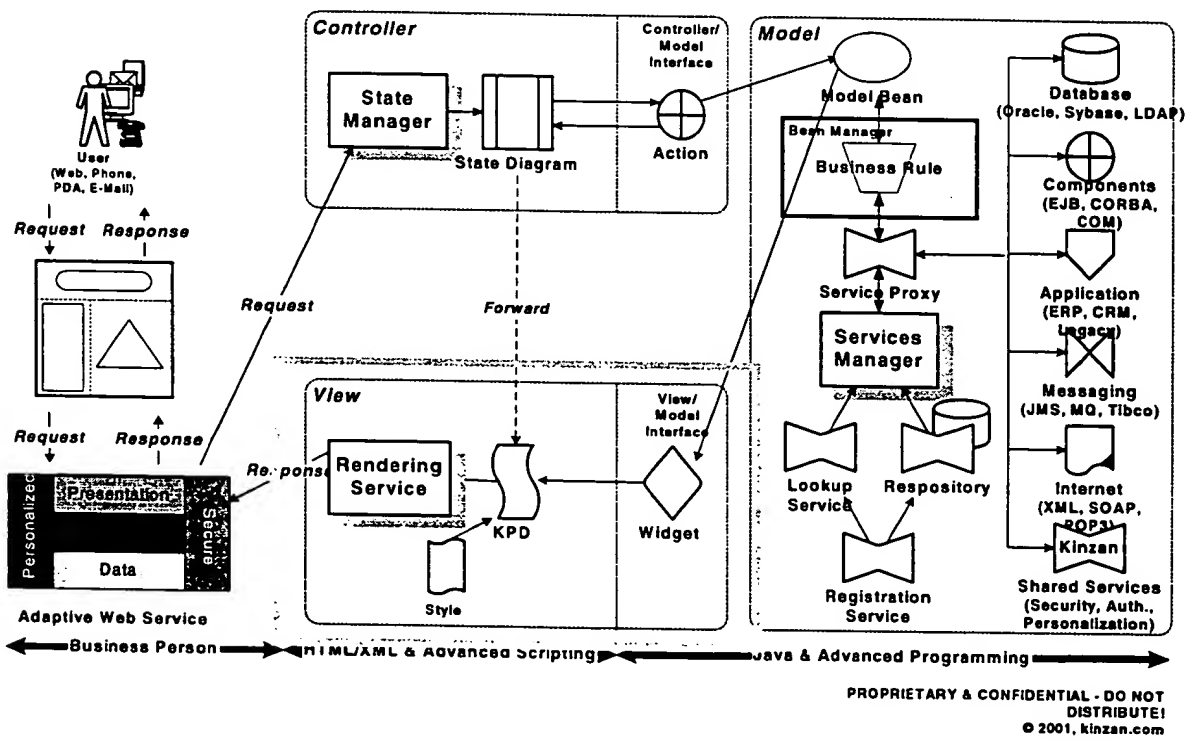
1.7.4 Using Existing Services

Once a service has been developed and deployed, it is available for use within the application. Most commonly, you will develop a widget that incorporates the service. For information on developing widgets, see section 1.8.3.

1.8 The View Layer

The classic definition of Model/View/Controller architecture describes the view layer as the visual representation of the state of the model to the user.

In the KTP architecture, the view layer consists of widgets, Kinzan Widget Templates (KWT files), styles, and assets. Widgets are used by components to format a visual representation, which is then placed on a page in the form of Kinzan Page Descriptor (KPD) definitions. The Rendering Service gathers information from all these sources to render and display the page.



1.8.1 The Elements of the View Layer

The elements of the view layer are responsible for rendering a contextually appropriate page and displaying it to the user. The view layer comprises the following main elements:

KINZAN

- Style file: Is an XML file that specifies style attributes, such as the fonts color, or table border size.
- Assets: Leaf objects to be included on a page, including images, text, HTML fragments, sounds, and so on. Assets may have variants that are dynamically resolved based on the current style, locale, or target device.
- The widget itself: A widget encapsulates a formatted and styled visual representation of data and has two parts:
    - Kinzan Widget Template (KWT) file: An XML file that governs the appearance of the widget. A KWT may have different variants based on device, locale, and/or style, and may contain generic zones for placing additional, nested components.
    - An XML definition of the widget, which includes declaring the widget's attributes and assets, as well as defining the name of the KWT to serve as the widget's template.

A widget is typically a generic view, which by itself has no meaning. For instance, an example widget may be an "image with a caption" widget, which displays an image with a text caption underneath. However, the actual image and text to display would yet be undefined. The component is responsible for binding meaningful data to the widget, as well as mapping children components to widget zones, which allows for nesting of visual components. This is done in the component definition, as discussed in Section 1.8.4.

In addition to the component, one more view layer element is required in order for the rendering service to build a page that can be viewed by the client, and that is the Kinzan Page Descriptor, or KPD. The KPD is an XML descriptor that specifies the title of the page, the root component to place on the page, and the style to apply to that component.

1.8.2 Widget Overview

As stated previously, a widget is nothing more than a generic view and requires bindings to data to make that view meaningful. Such bindings can be as simple as a literal text string, or may be more complicated, such as an attribute from an entity EJB. In either case, the widget definition and template remains simply centered on providing a view, and should not include any business logic specific to any application. In some cases, it may be necessary to develop a more specialized widget that presents a specific view, but for most applications, a set of generic widgets will be able to provide the needed elements to generate the view required.

1.8.3 Developing Widgets

Developing a widget typically involves:

- Declaring the widget in a Kinzan Application (KApp) file

- Creating a KWT file to govern the presentation of the widget

Skill Level:

Developing new widgets requires the same skills as in designing an HTML or WML page, depending on the desired target device. In more complex instances, some Java programming and JSP skill is needed.

Widgets

The first step in defining a new widget is to create a widget definition in the KApp file. The widget definition is an XML descriptor that lists the following items:

Widget Attributes: Widget attributes define holders for data that can be bound to and displayed in the widget template. An attribute needs to be defined for each data element that the template needs to access. Each attribute definition takes the form:

```
<ktp:local name="caption"
           type="java.lang.String"
           defaultValue="This is a caption"/>
```

where the name identifies the attribute, the type gives the java type of the attribute, and the optional defaultValue gives a default value to assign to the attribute if nothing is bound to it. This is useful when an a generic widget is defined where some attributes should be optional. For example, for an "Image with caption" widget and the above attribute definition, the component utilizing the widget may decide the default caption is satisfactory, and not bind another value to it.

Widget Assets: Widget assets map an asset defined elsewhere in the KApp file (See Section 1.8.6) to a local name, making the asset available to the widget template.

```
<ktp:widgetAsset name="myImage" asset="SampleAsset" />
```

The local name, given by the "name" attribute can be different from the asset name.

Widget Template: The widget template file that renders the view for this widget is also specified in the widget definition.

```
<ktp:widgetTemplate style="false" locale="false">
    ImageWithCaption.kwt
</ktp:widgetTemplate>
```

The widget template is loaded as an asset. Therefore, its definition is similar to an asset definition and includes the same attributes to signal the loader what types of variants exists. In this case, the definition defines that this template has no style or locale variations. By default, these attributes are "true".

This completes the definition of a widget. The next step is to define the widget template itself.

Widget Templates

A widget template file, or KWT, defines the visual presentation of the widget attributes and assets. KWT files are JSP fragments, allowing them to define more complex java code in the template to control how the page is rendered. However, they can also be simple HTML (or WML, or whatever markup language is required for the target device), augmented by special KTP tags that allow access to widget attributes and assets, as well as defining the appropriate forms for interfacing with the KTP controller layer.

Additionally, a KWT may have variants based on style, device, and locale. In this way, a component utilizing a given widget will automatically have views for the variant styles, devices, and locales that the widget supports. (Assuming that the other widgets and assets the component uses also support the same variations).

A sample KWT is given below.

```
1  <table>
2      <tr>
3          <td>
4              <ktp:image image="widget.myImage"/>
5          </td>
6      </tr>
7      <tr>
8          <td>
9              <ktp:text text="widget.caption"/>
10         </td>
11     </tr>
12 </table>
```

The template is for an HTML display of the image with caption example we have been discussing.

Line 4 Defines an image taken from the widget, named by the local widget asset name **myImage.**

Line 9 Defines some text to display, taken from the widget's **caption** attribute.

Widget Template Tags

As mentioned above, the KWT template is augmented by special KTP tags that are processed by the KApp Loader at compile time to generate the appropriate

JSP files. The KTP tags that are available to the KWT developer are described below:

```
<ktp:zone name="zoneName" />
```

This tag defines a zone in the widget by the given ***zoneName***. The component utilizing this widget is responsible for mapping content to the zone. The zone is simply a placeholder, and can be positioned by the template designer as desired.

```
<ktp:text text="value">
```

where ***value*** is either a text literal or *widget.**widgetAttributeName***. This will result in the text from the given widget attribute to be placed here, surrounded by styled font tags, if the device supports them.

```
<ktp:image image="image" border="b" width="w" height="h">
```

where image can be a literal or *widget.**widgetAssetName***. This will result in the appropriate asset to be displayed here, with the appropriate variation at runtime.

b, ***w***, and ***h*** can be literals or *widget.**widgetAttributeName***. These attributes are optional and may be ignored for certain devices.

```
<ktp:form>…</ktp:form>
```

This tag will generate the correct form to post to the controller layer, including hidden variables required for the State Manager to function, and JSP code to name it uniquely for a page. It is **not** recommended you try to manually code the form tags.

Inside the form, you can place:

```
<ktp:submit value="value" image="image">
```

where ***value*** is the value to post to the State Manager.

image is an optional attribute that if specified will result in the submit button being an image. If ***image*** is specified as *widget.**widgetAssetName***, then it will utilize the given asset. The javascript is generated automatically and keyed to the generated form name. This is not supported on all devices, and will be ignored.

Additionally, any other tag can take on new meaning:

```
<ktp:tagName attributes... >
```

where ***tagName*** can be any valid tag for the target device of the template. This will generate the stylized tag and search the attributes list for widget attributes. This is useful where you would like a certain attribute to be set from a widget attribute. For example:

```
<ktp:input type="text" value="widget.textInputValue"/>
```

would result in a text input box, where the value is set to the widget's *textInputValue* attribute.

Note that some of the tags require different formatting depending on the target device requirements. If a certain template is not functioning correctly, examine the output JSP for irregularities. It may be necessary to hand code the desired functionality.

More detailed widget examples and their usage can be found in Appendix A.

1.8.4 Components

The component is responsible for binding useful data to the widget. For example, the KApp snippet below identifies the **ImageWithCaption** widget as the view for the "display" mode, and binds some data from a literal to the widget's **caption** attribute.

```
1           <ktp:componentMode name="display" widget="ImageWithCaption">
2                   <ktp:bindings>
3                       <ktp:binding widgetAttribute="caption">
4                           <ktp:from literal="This is a literal binding!"/>
5                       </ktp:binding>
```

Line 1 Defines the component mode (i.e. display) and specifies the widget that will represent this component in this mode (image with caption)

Lines 3-4 Binds the literal `"This is a literal binding!"` to the widget attribute **caption**.

Bindings to widget attributes may also be from dissimilar data types, in which a transformation can be specified as an attribute to the **from** tag, which defines a data type transformation from the source of the binding to the correct data type of the widget attribute.

For a more in-depth explanation of components, see "The Component Overview" – Section 1.5 of this document. For more in depth examples, refer to Appendix A.

1.8.5 Developing and Using Style Files

The style file is an XML file that defines stylistic elements to apply to pages that reference the style. The format of the style file closely follows that used in Cascading Style Sheets (CSS), specifying a list of tags and the style attributes for each tag. Multiple classes may be defined in a style, allowing you to define variations of style attributes for given tags.

For example, a company wants to use its corporate colors on the home page for most of the year. However, in the spring and fall of each year it sponsors special events and wants the home page colors to change at these times. Defining separate classes in the `.style` file for Default, Spring, and Fall color schemes allows you to quickly and uniformly change the visual attributes for a page, and across multiple pages.

The Rendering Service applies the style transformation when the JSP file is compiled and executed at runtime. The style elements are stored separately from the JSP itself to allow runtime style configuration by the user, for example to allow the user to change the color scheme of the page.

Style transformations occur on the server, so you can benefit from CSS-like capabilities without depending on CSS-enabled browsers. Fonts are supplied by the client installation.

1.8.6 Managing Assets

Assets include objects such as image files, HTML snippets, PDF files, Flash movies, and sounds. Assets have variants that are resolved at run time based on locale, style, and/or target device.

Files that make up asset variants are stored in an `/assets` directory under the application's root, or somewhere else in the KApp loader's search path. Within the `/assets` directory, asset variants should be arranged hierarchically in `[locale]/[style]/[target]` order. The loader application is responsible for searching through these directories for valid variants and resolving them, so that at runtime, the rendering service can look up the required variant quickly. See section 1.6.1 for more information on the Kinzan site hierarchy.

Assets are declared in the KApp file. Their definition includes a description and a list of variant files, each variant optionally specifying the type of variant that exists. A sample asset declaration is given below.

```
1       <ktp:asset name="SampleAsset">
2           <ktp:description>This is an example asset</ktp:description>
3           <ktp:variant device="false"> image/sample.gif</ktp:variant>
4           <ktp:variant style="false" locale="false">image/sample.wbmp</ktp:variant>
5       </ktp:asset>
```

Line 1 Defines the asset and assigns it the name "SampleAsset"

Line 2 Gives a description to the asset. The description does not affect the rendering of the asset, but may be useful to developers or displayed in a tool.

Line 3-4 Defines possible variants for the asset. Note that while the loader application searches the file system for the actual variant files, the listing and attributes given help the loader to locate those variants. The device, style, and locale attributes of the variant tags signal the loader what types of variants exist. For this particular asset, the description describes two possible variant files that the loader will search for and associate with this asset. The first is named "image/sample.gif", for which style and locale variants exist, but not device. The second is "image/sample.wbmp", for which only device variants exist.

1.8.7 Kinzan Page Descriptors

The Kinzan Page Descriptor (KPD) is an XML descriptor that declares a page's title, style, and the root component that contains the page's content. A page's root component may be a simple leaf component, containing one widget, or a hierarchy of children components that form a complex view. The KPD definition is declared inside the KApp file. An example definition is given below.

```
1       <ktp:KPD name="samplePage" title="Sample Page">
2           <ktp:description>My sample page.</ktp:description>
3           <ktp:component name="sampleComponent"
4                          type="ImageWithCaption" />
5       </ktp:KPD>
```

Line 1 Declares the KPD, giving it a name and title. The name is the handle that the rendering service uses to look up the page.

Line 2 A description is given to the page. This does not affect the rendering of the page, but is data that may be used by tools so developers have a more meaningful description of the page.

Lines 3-4 The component instance for the page is defined. In this case, the page is defining an instance of the component type **ImageWithCaption**, and naming it **sampleComponent**. The page may also reference an existing component instance.

1.8.8 Rendering a Page

The Rendering Service is responsible for dynamically assembling KPD's when requests are sent to the web server. This is done by looking up the root component of the page, and then building it and any children it may have.

The Rendering Service is fully dynamic, and may thus (with proper tools) be manipulated and configured on the fly, without having to recompile site pages or site modules. This rapid configurability is a key feature of the KTP Framework.

The Rendering Service internally uses the Model-View-Controller (MVC) paradigm to render pages.

The Model is the set of Java classes supplied with the widgets. These Java classes are responsible for creating the widgets, retrieving any data required by them and implementing any business logic. Note that internal to the rendering environment, all components (pages, widgets, containers, etc.) are modeled as widgets.

The View is the visual representation of the widget. In the rendering system, this is embodied in the corresponding widget template KWT file, which is processed into a JSP file for the widget. When the processed KWT file is compiled into a servlet and executed, the servlet interrogates the widget and uses the data contained in the widget to create a visual representation of it.

The Controller is the rendering servlet. The rendering servlet processes requests by matching a request to a model and view.

Rendering a page involves several components. These are described below:

Rendering Servlet: The rendering servlet processes requests to render pages. It forwards the request to the rendering service and invokes the appropriate servlet/JSP for the page.

Runtime Persistence Service: The Runtime Persistence service encapsulates all operations for the Runtime persistent storage as a service, where the page model is stored. The KApp loader application is responsible for loading the required data to describe the components, widgets, styles, and pages into persistent storage at compile time.

Rendering Service: The rendering service is responsible for building the model of a page. It uses the Runtime Persistence service to retrieve data from the Runtime datastore. For each component on the page, the rendering service uses this data to create the corresponding widget. The page model is returned as a tree structure where the root of the tree is the page widget.

Component Service: The component service is responsible for actually building the components on the page, performing the required bindings, and setting the appropriate widget for the component's mode.

Widget: The widget object contains all the data required to render the widget. At runtime, a widget also provides access to all its defined attributes.

Device Template: The device template is the main wrapper JSP for a given target device. The rendering service forwards to this JSP when it is done

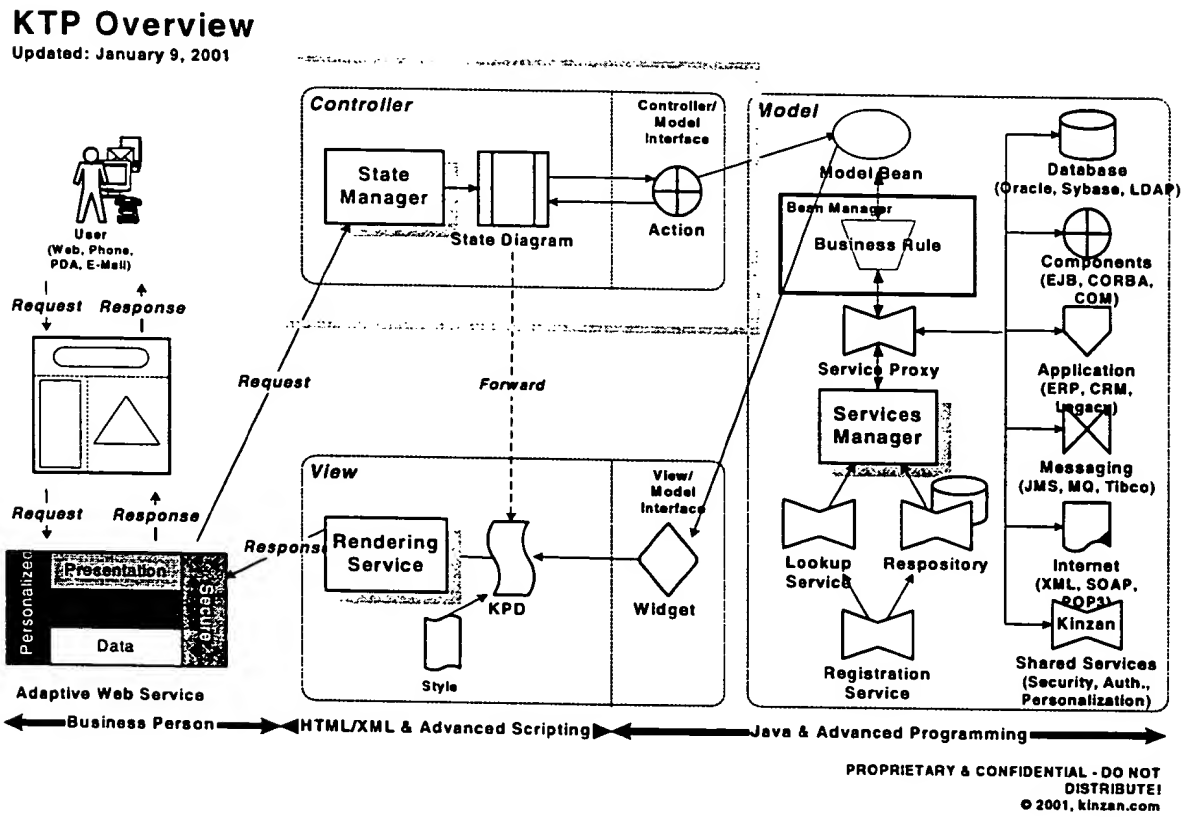
building the components for the page. The device template includes the main root component of the page by including the template for the widget in the component's current mode, which was previously resolved by the component service when it built the component. This template, in turn, will include any children components in the same way.

Widget Template KWT: The widget template KWT (once processed and compiled into a servlet) translates the widget attribute data that was bound to it into its visual representation. It may also include assets, and applies styling if required

1.9 The Controller Layer

The classic definition of Model/View/Controller architecture describes the controller layer as the means by which the user provides input or changes the state of the model.

In the KTP architecture, the controller layer consists of the State Manager, Kinzan State Diagrams (KSD files), and Action files.



1.9.1 Kinzan State Manager Overview

The Kinzan State Manager is a powerful and flexible tool for assembling web applications. Using the State Manager, you can rapidly assemble web pages, typically Kinzan Page Descriptors (KPD's), and application logic to configure your application.

The State Manager supports flexible application configuration by cleanly separating state management from the presentation layer and providing a

mechanism to chain application logic modules to connect application pages. It also provides for state management at the user and request level, avoiding many concurrency issues that plague less sophisticated state management systems (including duplicate request handling).

Application logic is implemented in modules using Java objects called actions. These modules allow for "pluggable" components that you can assemble dynamically based on user input and the current state. Developing atomic components and chaining them together provides flexibility in modifying or customizing applications for different users. The State Manager also enables convenient and extremely powerful field validation by these components.

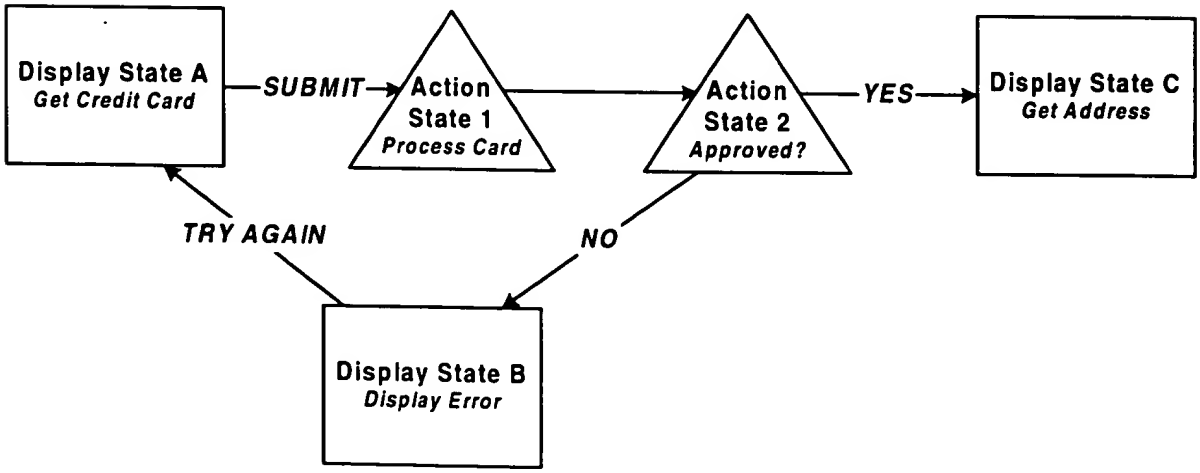
The State Manager provides basic transaction management across actions, and makes it easy to link to business logic (usually Enterprise Java Beans (EJB's)) and various support services.

Applications can have many mini-applications (or subroutines) within them, allowing for better logical segmentation of application functions. This segmentation encourages state diagram reuse within and between applications, streamlining application development. The State Manager supports coordination and handoff between state diagrams.

Kinzan State Diagrams (KSD's) are defined using XML, allowing for flexible configuration of wizards by less sophisticated developers, with easy integration into support tools.

1.9.2 What Is a State Diagram?

A state diagram, represents discrete steps in an application flow with decision points that trigger transitions to different sections of the application.



KINZAN

For example, in the figure above, the application begins in Display State A. When the user enters a credit card number and clicks the submit button on a form to generate a SUBMIT event, the application moves to Action State 1. Action State 1 processes the credit card number and forwards the result to Action State 2. Action State 2 applies business rules to the credit card processing result to determine whether the merchant will accept the credit card.

If the rules in Action State 2 generate a YES event, control passes to Display State C, which presents the user with a form on which to enter an address. If the rules in Action State 2 generate a NO event, control passes to Display State B, which presents the user with a rejection message and provides the opportunity to enter another credit card number.

In Display State B, if the user chooses to enter another credit card number (perhaps by clicking a Try Again link), a TRY AGAIN event is generated and control passes back to Display State A for the process to begin again.

1.9.3 Advantages of Using a State Diagram

HTTP is a stateless protocol. Once a web server transfers a web page to a web browser, it has no knowledge about how pages are related to one another. This is one reason why links are embedded within web pages instead of being managed by the server.

You can use a state manager and state diagrams to describe sophisticated relationships within an application without needing to manage all sorts of tricks to manipulate the actual web pages.

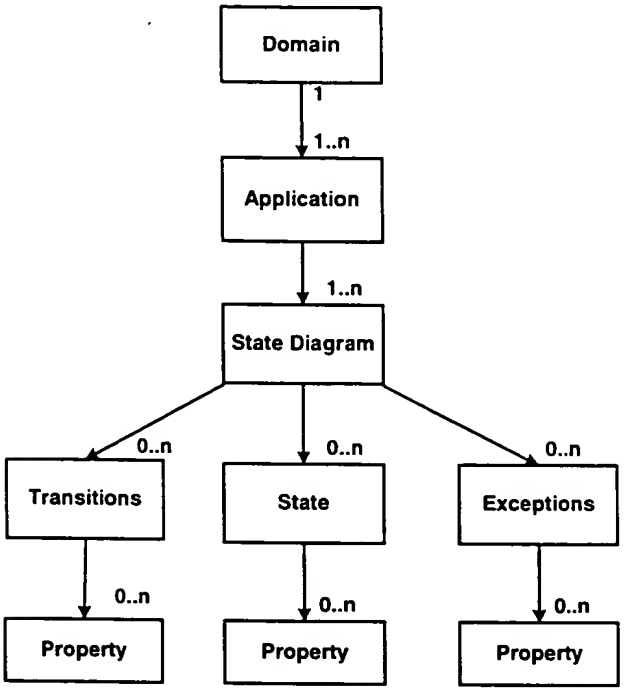
When you model applications using traditional state diagrams, you immediately benefit from enhanced documentation, manageability, flexibility, and reuse across applications.

Using the example in the previous section, you could easily replace Action State 2 with a different piece of business logic that changes the criteria used to approve or deny a credit card transaction. You could make the change by replacing a single component, rather than having to dig through a much larger piece of application logic distributed throughout presentation code.

If, in the future, the application needs to be more sophisticated to support multiple approval components for different merchants, modifying the application would be straightforward. You could insert a piece of application logic between Action State 1 and Action State 2 to determine which of many possible approval components to invoke for a particular merchant and trigger that approval component, all without impacting the other components of the application (logic or presentation).

1.9.4 Components of the Kinzan State Manager

The following figure represents the various components of the State Manager model:



Domain

A domain is a collection of applications. All state requests should be made through this interface in order to support the inheritance of applications, state diagram, etc. between parent and child sites.

Application

An application is a collection of associated state diagrams, which can by themselves be thought of as mini-applications. Creating associations of state diagram provides a common deployment group for a particular application. You can define an application hierarchy, assigning parent-child relationships to applications. A child application inherits state diagrams, states, and transitions from its parent application.

State Diagram

A state diagram, is a set of states, events, and pointers to actions and pages. An example would be a Tax State Diagram or a Shipping State Diagram.

Stat

A state is a location in the state diagram being managed by the State Manager.

States may be either display states, which reference a KPD or URL, or action states, which reference an action for processing.

Transitions

States may have zero to many transitions to other states, which are triggered by various events.

Exceptions

Exceptions define catch statements for state diagrams. When an exception occurs during the execution of a state diagram, it may be caught and transfer control to another state.

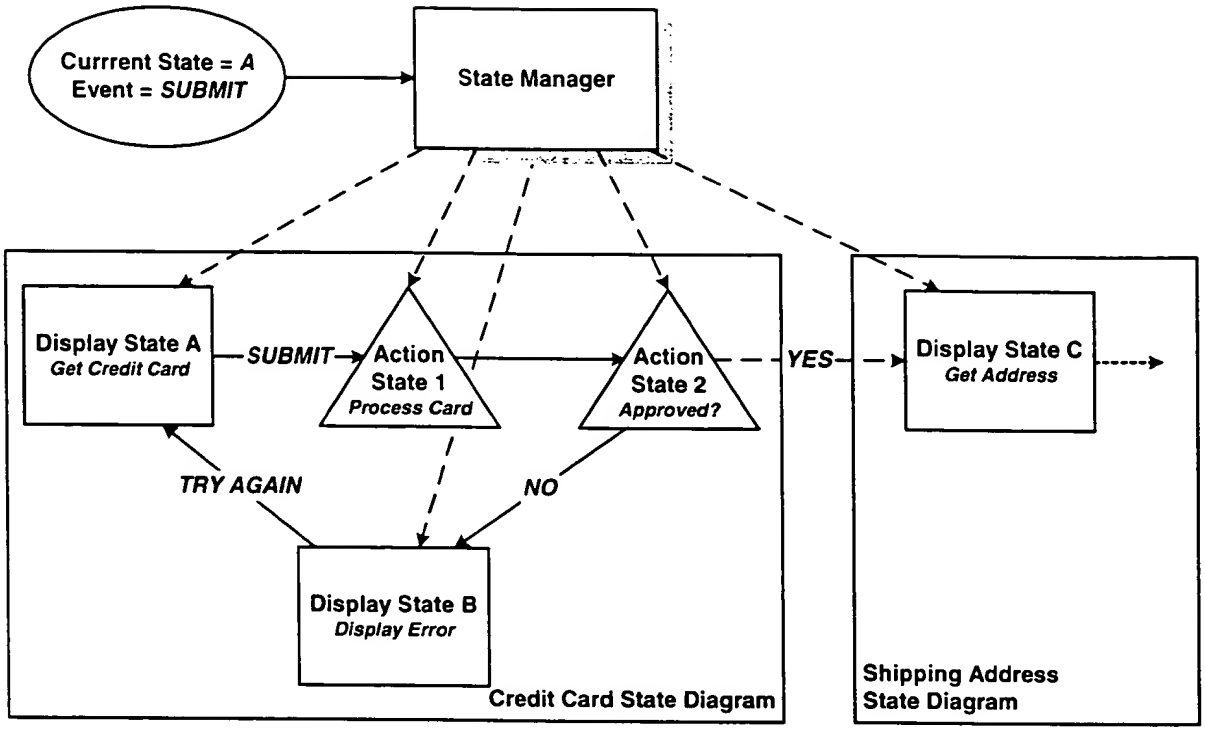
Properties

States may also have properties associated with them as name-value pairs. This may be useful in configuring a shared piece of application logic (or Action State) for a particular application.

1.9.5 State Manager Details

State Manager

The State Manager controls the execution of a domain. Based on a current state and an input event, it retrieves the next `State` object. If the state is an action state, the associated code executes using the current `UserContext` as input. The State Manager processes the resulting event and retrieves the next `State` object. This loop continues until a display state is triggered, at which point control returns to the calling code.



For example, in the above figure an ACTION from a web page, which passes some form variables with a SUBMIT event, invokes the State Manager. Because the application is in Display State A, the State Manager queries the transitions for Display State A to determine that control passes to Action State 1 on a SUBMIT event. Action State 1 retrieves the form variables from the request context and executes its application logic.

The State Manager manages all state diagrams in an application. In the above example, a YES event from Action State 2 passes control to the entry point of the Shipping Address Wizard.

1.9.6 State Types

States may be associated with a display page or with an action.

If the state is associated with a display page, the page is returned to the user and the State Manager waits for input. One output from this page is an event, which the State Manager uses to determine the next state.

If the state is associated with an action, the action executes when the state is entered. The code returns an event, which the State Manager uses to determine the next state.

DisplayState

The `DisplayState` class represents a state with an associated user-interaction page. The user-interaction page is referred to as either a KPD (using the `kpd` attribute) or as a redirect URL (using the `forward` attribute). Only display states are stored in a user's state history.

ComponentState

Is the component's display mode.

ActionState

The `ActionState` class represents a state with an associated action. It contains a reference to an action, which is the interface for defining actions.

`ActionStates` return events that signify what they have done. For example, a `ProcessOrder` component might return `OrderProcessed` or `OrderFailed`. In addition, the Action can throw an exception to signify that an error has occurred.

An `ActionState` may also accept input and make a decision about the next appropriate output state. A simple example would be an `ActionState` with the reseller ID of the user as its output event. You could use the reseller ID as an event to trigger the next state based on the ID of that reseller.

1.9.7 Action

The action interface defines a single piece of code that is associated with a state. Its main method receives a context (which includes session and request information), performs an action, and returns an output event. This output event leads to another state, which may be either another `ActionState` or a `DisplayState`.

1.9.8 State Servlet

The State servlet manages retrieval of the parameters from the form. It places the parameters into an object and passes them to the State Manager for processing. When the call to the State Manager returns, the State servlet dispatches a request to the resulting KPD file, or redirects to a new URL, depending on what was returned by the State Manager.

1.9.9 Typical Sequence of Events

In a typical use of the State Manager, the action from an HTML form invokes the State servlet, passing in the various form parameters and session information. The State servlet packages the input and passes it to the State Manager.

Note: The term "wizard" is used in the source code to refer to a state diagram.

The State Manager retrieves the request. It expects to see parameters named `net_kinzan_nextWizard`, `net_kinzan_nextState`, and `net_kinzan_nextEvent`. These constants should be pulled from the `net.kinzan.state.StateData.java` file and represent constants. It determines the next state by using the appropriate `Domain`, `Wizard`, and `State` classes, which may be different than current wizard and current state.

Alternatively, if the `stateDate.REQUEST_CONTEXT_SEQ_ID` parameter is set to the user context sequence ID of the form, the `net_kinzan_nextWizard` and `net_kinzan_nextState` parameters are assumed to be the same as the current wizard and current state. The `net_kinzan_nextEvent` parameter still determines the next event.

If the next state is an `ActionState`, the state's action is retrieved and executed. The resulting event is then used to retrieve the next state, which is then similarly processed.

If the next state is a `DisplayState`, control returns to the caller, with the `DisplayState` as the return value. The State servlet then dispatches control to the appropriate KPD file (the display state defines a `kpd` attribute), or redirects to a new URL (if the display state defines a `forward` attribute).

1.9.10 State Diagrams as Independent Mini-Applications

One of the major design abstractions when using the State Manager is the state diagram. A state diagram is a set of states that act as an independent mini-application. Encapsulating these mini-applications encourages reuse within and between applications. For example, a Tax State Diagram for configuring tax rate may be accessible either from a store-install state diagram or from a central administration page.

Decomposing an application into many mini-applications is useful and powerful, but it does create the need to coordinate control between state diagrams. For example, if you use a Tax State Diagram in multiple areas of a web site, how does the Tax State Diagram know where to return when it finishes? The Tax State Diagram should not need to know what other states or pages might link to it and you should not need to hardcode the return state. Having to do so would negate many of the advantages of encapsulation.

The State Manager automatically coordinates interaction between state diagrams by maintaining a call stack that tracks state diagram START and END states . When a state diagram reaches its own END state, the State Manager returns

control to the state that called the state diagram. Alternatively, the state that called the state diagram may give the State Manager an explicit return location. To use this feature without needing to hardcode a state diagram's return state, transition to `#END`, when you want to exit your state diagram.

The State Manager determines the return location for a state diagram in one of two ways. If a request to that State Diagram is made with the parameter `StateData.REQUEST_RETURN.URL`, the value of this parameter becomes the return location of the wizard. If no such parameter exists, the state prior to entering the new state diagram is set as the return location of the state diagram.

In either case, when the `#END` state is reached, the return location is retrieved by the `EndStateAction`, which directs the State Manager to move to that location in the state diagram.

Below is a list of classes used to implement this feature, along with descriptions.

EndStateAction

The `EndStateAction` is an `Action` that redirects the application to the state diagram's return location. To do this, it checks for a return location in the `ReturnStack`. If it finds one, it throws a `JumpToStateException` constructed with the return location. The State Manager catches the `JumpToStateException`, extracts the return location from it, and directs the application to that state. The EndStateAction in TP4 is built into the State Diagram by a transition to `#END`.

ReturnStack

The `ReturnStack` encapsulates a stack of return locations for an application session. It is important to have a stack (rather than a single variable) because State Diagrams may call other State Diagrams, and each has a different return location. Without a stack the return location would be overridden with each new transition.

JumpToStateException

A `JumpToStateException` directs the State Manager to move to a new state, circumventing the event-driven state transition model. An `Action` may throw this exception whenever it needs to move to a state that cannot be reached by any transition.

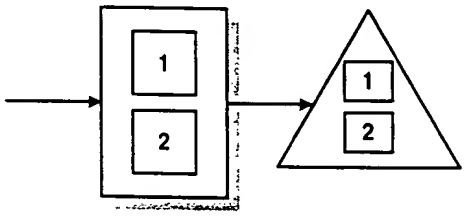
1.9.11 Usage Scenarios

This section explores the kinds of things that you can do with the State Manager. It examines various problem scenarios and how to apply the State Manager to solve these problems. In the process, we introduce some advanced features of the State Manager.

1.9.11.1 Overloaded Form Processor

This example looks at a common scenario when doing server-side processing of form data. Consider the case where a single form returns two addresses: one for receiving payments and one for the physical store location. Hitting submit on this form returns control to the server, which retrieves form variables from the page and updates the addresses in the database before redirecting to another page.

The following diagram depicts this situation. One HTML page has essentially two separate sets of information in its form. It submits to the server, which then executes two address updates using a common processor.

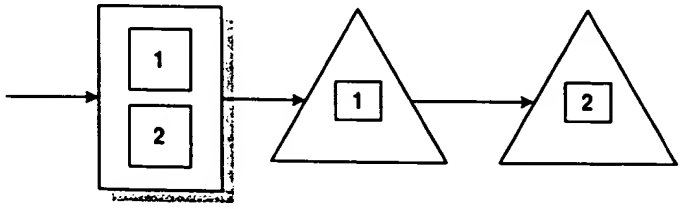


Overloaded Form Processor

But what happens if a new reseller decides it wants to collect and update the payment and physical addresses on separate forms? The obvious solution is to write two new form-processing scripts and two new HTML pages to get this functionality. This is tedious, error-prone, and makes maintenance and upgrades more difficult.

The State Manager allows you to redesign this part of the application. Rather than having a single form processor for all the information on the form, you can write two individual actions to perform the two distinct database updates. Once you divide the form processor code into two parts, you have a lot more flexibility.

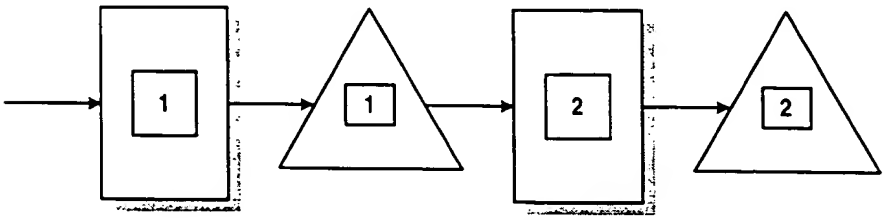
One option is to keep the same user interface. One HTML page has the form parameters needed for the two components. Submitting the form on the HTML page moves to the first action state, which performs its database update and then sends execution to the second action state.



Separate Components

Using the same code components, we can now easily meet our reseller's request to divide the UI into separate forms.

Here is another reseller scenario. The following diagram depicts form 1 (payment address) posting to the action for updating the payment address. After this, the user is directed to the second form (physical address). A submit from that form then calls the action for updating the physical address.



Separate Components, Separate Forms

The State Manager enables this new level of flexibility. This example shows how a more modular, component-based system allows user interfaces to be more modular as well. Specifically, HTML forms that drive specific updates may be composed into a single page, or pulled apart into separate pages, while still sharing common application logic.

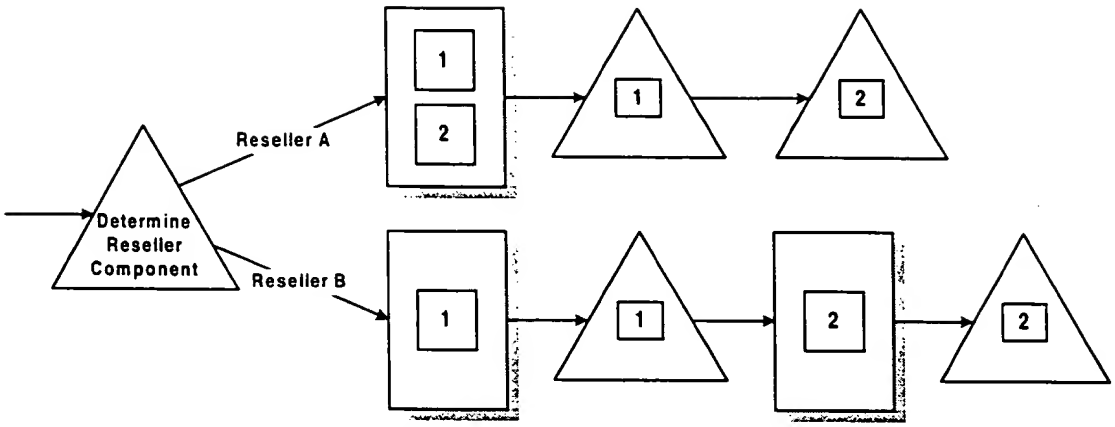
Application Needs to Change Based on Information Not Embedded In the State Diagram

By allowing actions to return an event, which may then be used to drive the next state, the State Manager allows application flow to be affected by internal logic. Although most actions will probably either succeed or fail and proceed to the corresponding success or failure state, this is not a constraint. Actions may also direct the State Manager by returning any kind of event.

Consider an example where two resellers share the same deployment of an application and require the application to behave differently. Using the address form situation from the first example, Reseller A wants a single form to drive both database updates, but Reseller B wants two distinct forms.

The State Manager provides a straightforward solution to this situation. First, create a new action that has the Reseller ID of the current user as its output event. Then, use this component to drive the application to either of the alternatives.

The following diagram depicts this situation. The Determine Reseller component is the first action state to which the user is sent. The Reseller ID, the output of the component, drives the application to the appropriate variant. Each variant uses common application logic, with changes constrained to the presentation layer.



Reseller Driving Application Variations

Determining the reseller is just one example of how to control the application dynamically. Logical decision components can be added any time you need to change the application dynamically.

1.9.11.2 Duplicate Request Handling

Duplicate requests can be a major problem for web applications. The State Manager provides elegant duplicate request handling in a transparent way.

What Is a Duplicate Request?

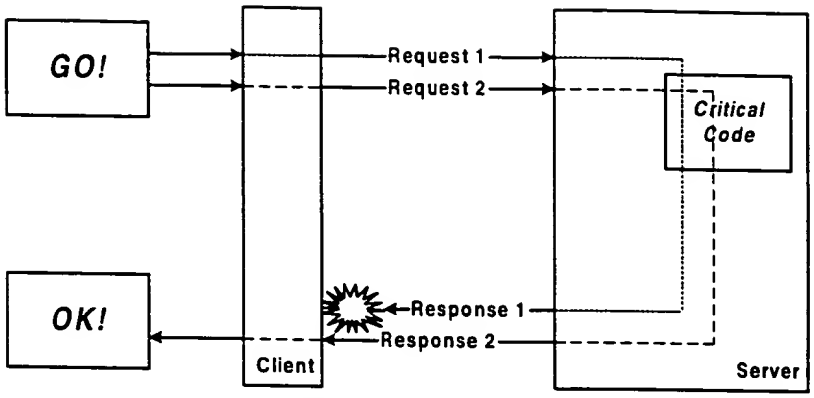
A duplicate request occurs when a user clicks on a web page submit button more than once before the server returns a response, sending the same request to the browser multiple times. The client ignores the response of the first request and displays the response of the last request.

Duplicate requests can be quite common. A user who clicks a button and does not perceive any change may grow impatient and click the button again. With web-based applications, this may happen quite often.

Why Are Duplicate Requests Bad?

Typically server processes are not programmed to identify duplicate requests, so the server processes both requests, perhaps attempting to perform some critical action (such as charging a credit card) more than once.

Here is a picture of a duplicate request in terms of client-server interaction:



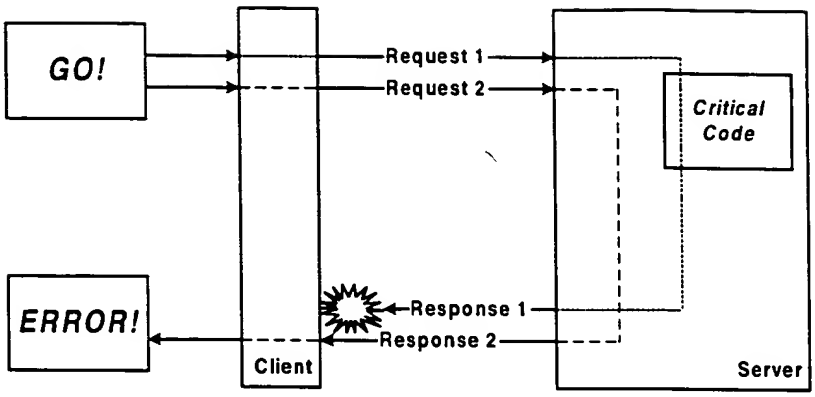
Duplicate Request: No Handling

Request 1 and Request 2 are identical and occur within milliseconds of each other—the result of a user double-clicking a submit button on a form. The server handles both requests, and both requests generate responses. The response for Request 1 does not make it to the client because the client is not listening for it. Instead, the client listens only for the response to Request 2, which it does receive. Both requests executed the critical code, which might make database tables inconsistent, charge a credit card twice, or cause other damaging side effects.

Typical Duplicate Request Handling

The above scenario, no duplicate request handling, is commonplace in web applications. To address the problem, some web application developers write specific functionality for duplicate-request handling into their server code. Typically this code checks for two requests made by the same user within a very small amount of time. If the server detects a duplicate request according to these parameters, it stops processing the duplicate request and returns an error message to the user.

Here is a diagram of typical duplicate-request handling with more sophisticated web applications:



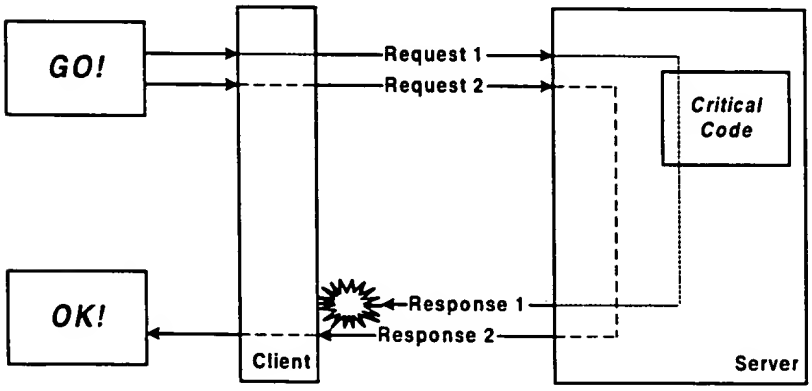
Duplicate Request: Common Handling

In this case, the server has been safeguarded from the harm of a duplicate request because the duplicate request does not reach the critical code. It is, in effect, turned away at the door.

This solution has its own issues though. The client receives only an error message informing him that he made a duplicate request. The server must return an error message, rather than the appropriate response, because the first and second requests were processed in two separate, disconnected threads. The second thread has no way of returning the response of the first request to the client, which would be ideal because the client ignores the response from the first request.

Kinzan State Manager Duplicate Request Handling

The State Machine architecture more properly handles duplicate requests. Because the State Manager maintains context information at the page and response levels, it can accurately detect duplicate requests and return the appropriate output to the client.



Duplicate Request: Handled Properly by the State Machine

The State Manager can accurately determine if a request is a duplicate because a `UserContext` object is saved with each request. The `UserContext` contains all of the form variables and values that belong to the request. A new request can be compared against the previous one, value by value.

This is a great advantage over many duplicate-request determination algorithms, which simply look for two requests coming from the same user within a very small amount of time (usually milliseconds), and which do not consider the actual request values. These algorithms are only useful in handling the accidental "nervous twitch" double-click; they do not behave appropriately with the more common situation where a user is uncertain that the button click was accepted and decides to click it again several seconds later.

The State Manager never actually produces any output; it redirects or forwards the user to a new location whenever processing reaches a display state. When a duplicate request is detected, the second request can find the display state that is the result of the first request (the one the user really wants to see) and redirect the client to that page.

If the first request is not finished when the second request tries to find the result display state, the State Manager redirects the client to a temporary "Please Wait" page. This page refreshes automatically, checking the server again to see if the first request has finished. If it has, the State Manager will return the result state, otherwise it will return the "Please Wait" page again.

The result is a very elegant and natural handling of duplicate requests. The server and user are safeguarded from the harm of duplicate requests, the user is advised of double-clicks (potentially avoiding future duplicate requests), and the user gets the appropriate response when the original request finishes.

How do I setup my application to handle duplicate requests?

When the State Manager detects a duplicate request, it throws a DuplicateRequestException. Your application state diagram should catch this exception and transition to a "Please Wait" page. Here is the state diagram XML to do this:

```
    <ktp:catch exception="net.kinzan.state.DuplicateRequestException"
state="pleaseWait"/>
    <ktp:displayState name="pleaseWait" forward="/test/pleaseWait.jsp"/>
```

The "Please Wait" page should give the user a message that the request is being processed. This "Please Wait" page has one requirement--that it generates an event named "WaitForDuplicateRequest". This instructs the State Manager to wait for the initial request to finish processing and, when it does finish, the State Manager will return the appropriate results to the user.

1.10 Logging Service

The `LoggingService` integrates the IBM JLog package into the KTP environment and implements two loggers: a trace logger and a message logger.

- The trace logger is intended to be used by KTP developers as a debugging tool.
- The message logger is intended to send messages to users of Kinzan-developed systems, such as system administration personnel and web page designers, and is capable of being localized.

The trace logger and the message logger can each be run at the system level and at the application level.

1.10.1 System Loggers

System loggers are visible across multiple KinzanApp instances. To access a system logger, use the `GetSystemTraceLogger` and `GetSystemMessageLogger` static methods.

`net/kinzan/logging/LoggingService.properties` stores system logger properties. If the system logger cannot locate this file, it defaults to full synchronized logging and sends output to the console. However, if the file exists, the output destination must be specified explicitly.

1.10.2 Application Loggers

Application loggers are created by a specific instance of the LoggingService, therefore they are only visible within a KinzanApp. They can be accessed in the usual way:

```
AppContext appContext = KinzanApp.getContext()
Logger traceLogger = appContext.getTraceLogger();
```

Use the application loggers whenever possible. They provide greater flexibility in that you can have different settings for each application logger.

1.10.3 Logging Output

You can configure each logger in the service to send its output to one or more supported devices. Currently supported devices include the console, files, and sockets. For each output device, you can configure its filter and message formatter.

1.10.4 Setting Properties

The logging service supports the following properties:

`Logger.Description` (optional)
A brief description that identifies the logger.

`Logger.Organization, Logger.Product, Logger.Component, Logger.Server, Logger.Client` (optional)
Used by the formatters to assemble a line in the logger.

`Logger.Message.MessageFile`
Sets the path to the resource bundle the message logger should use.

`Logger.Trace.Logging, Logger.Message.Logging`
When the appropriate property is set to `true`, turns on trace or message logging.

`Logger.Trace.Synchronous, Logger.Message.Synchronous` (optional)
When set to `true`, the call to the Logger object becomes a blocking call. During development, you can set these to `true`, but in production, these values should be set to `false`. The default is `false`.

`Logger.Trace.Console, Logger.Message.Console`
When set to `true`, directs logger output to the console. The default is `false`.

`Logger.Trace.Console.Formatter, Logger.Message.Console.Formatter` (optional)
If `Logger.Trace.Console` or `Logger.Message.Console` are set to `true`, these fields specify the class used to format a message. Valid options are:

- `com.ibm.logging.EnhancedFormatter`
- `com.ibm.logging.EnhancedTraceFormatter`
- `com.ibm.logging.TraceFormatter`
- `net.kinzan.logging.SingleLineTraceFormatter`

`Logger.Trace.Console.Mask, Logger.Message.Console.Mask`
Use these fields to set the message types that are allowed through the filter for the Console device. For instance, if you specified: `TYPE_EXIT TYPE_LEVEL1 TYPE_LEVEL3` for the Trace logger, only messages from the Trace logger with type `EXIT`, `LEVEL1`, and `LEVEL3` are displayed on the console. For valid type values, see the `IRecordType` interface.

`Logger.Trace.File, Logger.Message.File` (optional)
When set to `true`, directs logger output to a file.

Logger.Trace.File.FileName, Logger.Message.File.FileName
If `Logger.Trace.File` or `Logger.Message.File` are set to `true`, these fields specify the filename to which the logger writes messages. Do not use backslashes (\ ) in your path, the JLog package does not like them.

Logger.Trace.File.MaxFileSize, Logger.Message.File.MaxFileSize
If `Logger.Trace.File` or `Logger.Message.File` are set to `true`, these fields specify the maximum file size in KB. Once the file reaches this size, the logger creates a new one. The default is 1MB.

When the system creates the first new file, it appends 1 to the original filename; it appends n+1 to each succeeding file (so `myfile.log` becomes `myfile1.log`, `myfile2.log`, etc.).

Logger.Trace.File.MaxNoFiles, Logger.Message.File.MaxNoFiles
If `Logger.Trace.File` or `Logger.Message.File` are set to `true`, these fields specify the maximum number of files that the logger should keep when it rotates the log files.

Logger.Trace.File.Formatter, Logger.Message.File.Formatter
See `Logger.Trace.Console.Formatter`

Logger.Trace.File.Mask,Logger.Message.File.Mask
See `Logger.Trace.Console.Mask`

Logger.Trace.Socket, Logger.Message.Socket
When set to `true`, the system directs log output to another machine. See the JLog documentation for instructions on setting up the server daemon that should listen to these messages.

Logger.Trace.Socket.ServerName,Logger.Message.Socket.ServerName
When `Logger.Trace.Socket` or `Logger.Message.Socket` are set to `true`, these fields specify the server that messages are routed to.

Logger.Trace.Socket.Port,Logger.Message.Socket.Port
When `Logger.Trace.Socket` or `Logger.Message.Socket` are set to `true`, these fields specify the port numbers that the logging daemon is listening to.

Logger.Trace.Socket.Formatter,Logger.Message.Socket.Formatter
Please see `Logger.Trace.Console.Formatter`

Logger.Trace.Socket.Mask,Logger.Message.Socket.Mask
Please see `Logger.Trace.Console.Mask`

More information on these settings can be found in the JLog javadoc.

1.10.5 Using the Logging Service

`StateManagerServlet`, `BasicLogComponent`, `RequestContext` and `StateEngine` can now use the LoggingService.

The KinzanApp object can be used to return the AppContext and thereby returns Logger objects. This is the Logger object defined in the IBM JLog package.

Here is an example on how to use the class:

```
// get loggers
Logger traceLogger = KizanApp.getContext().getTraceLogger();
Logger messageLogger = KinzanApp.getContext().getMessageLogger();
traceLogger.text( IRecordType.TYPE_LEVEL1, "MyClassName", "MyMethodName", "Hi MOM!" );
```

- The standard name for the logging service is `LoggingService`.
- Constant classes, such as `IRecordType`, are not implemented.
- By convention, logger names are `iTraceLogger` and `iMessageLogger`.

> Be sure to use the correct record type when you log entry and exit messages. This makes it easy to turn them on and off later. For exit logs, use `IRecordType.TYPE_EXIT` and for entry use `IRecordType.TYPE_ENTRY`.

1.10.6 Filters

`AttributeFilter` allows you filter any attribute on a `LogRecord`, either through inclusion or exclusion.

The following JLog attributes are standard:

- `loggingClass`
- `loggingMethod`
- `organization`
- `product`
- `component`
- `client`
- `server`

The first two, `loggingClass` and `loggingMethod`, are typically the values you pass in to `logger.entry`, `logger.exit`, `logger.text`, `logger.exception`, etc. (The other five are typically configured on a per-LoggingService basis. They will become more important as we start deploying multiple web applications.)

Using `loggingClass` and `loggingMethod`, you can include only specific classes and/or methods you want to see and filter out all others or the opposite, exclude specific classes and/or methods.

The filter setting is in the default properties for both `GlobalSystem` and `LoggingService`. They have been commented out, so the typical behavior is still in play. But if you uncomment the line below for desired `LoggerName` and `LoggerType`, you will activate the `attributeFilter`.

```
#Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute=loggingClass loggingMethod
organization product component client server
```

You can either exclude the listed attribute values or include them:

```
Logger.{LoggerName}.{Console|File|Socket}.Filter.match=include
```

or

```
Logger.{LoggerName}.{Console|File|Socket}.Filter.match=exclude
```

You can also specify an Any or All filter by setting the `matchAll` value - match one value of each attribute (either all attributes or just one). To match all attributes listed, use `true` or to match any attribute listed use `false`.

```
Logger.{LoggerName}.{Console|File|Socket}.Filter.matchAll=true
```

For each attribute you can specify a space-separated list of values as an include or exclude list that must be matched by the logger.

```
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.loggingClass=
                                    net.kinzan.security.auth.module.JndiLoginModule
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.logginMethod=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.organization=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.product=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.component=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.client=
Logger.{LoggerName}.{Console|File|Socket}.Filter.attribute.server=
```

1.10.7 Additional Options

You can define more than one logging service if you need to. The biggest disadvantage with this approach is that not all classes have access to a request context. In these situations, the code to get the Logging service becomes convoluted:

```
AppContext appContext = KinzanApp.GetAppContext( "MyAppName", null );
LoggingService loggingService = KinzanApp.GetService( appContext, "MyFavoriteLogger" );
```

This implies that you have access to the Application instance name and the Logging service name.

Appendix A KTP Examples

1.TextBox Component

1.11 Introduction

1.11.1 Audience

This document is intended for developers who will be working with the Kinzan Technology Platform. It explains how to use the TextBox component.

1.11.2 Overview

This document describes the TextBox widget. This widget displays a text/password input file.

1.12 TextBox Component

1.12.1 TextBox Widget Definition

The GenericDataTable widget is defined in the file "simplegui.kapp". Here is the excerpt containing the widget definition.

```
1     <ktp:widget name="TextBox">
2               <ktp:attributeList>
3                       <ktp:local name="name" type="java.lang.String"
              defaultValue="TextBox"/>
4                       <ktp:local name="isPassword" type="java.lang.String"
              defaultValue="N"/>
5                       <ktp:local name="size" type="java.lang.String"/>
6                       <ktp:local name="maxlength" type="java.lang.String"/>
7                       <ktp:local name="value" type="java.lang.String" defaultValue=""/>
8               </ktp:attributeList>
9               <ktp:widgetTemplate style="false"
              locale="false">GenericTextBox.kwt</ktp:widgetTemplate>
10    </ktp:widget>
```

Line 1 The widget is defined with the name "TextBox"

Lines 2-8 The attributes of the widget are defined and given default values (more below).

Line 3 This attribute specifies the name of the text/password input field.

Line 4 This attribute specifies whether this component is a password field ("Y") or not ("N").

Line 5 This attribute specifies the display size of the text/password field.

Line 6 This attribute specifies the maximum allowed length of input to this field.

Line 7 This attribute specifies the current value of the contents of the input field.

Lines 9 The widget template. This points to the widget template file. There are no style or locale variants for this widget. However, there are device variants.

1.12.2 Device Variations

The widget supports variants in html and WML. This is achieved by defining the appropriate variant templates for each of the widgets utilized in rendering the display for the component.

file: simplegui.kapp

```
<?xml version="1.0"?>
<!DOCTYPE ktp:kapp SYSTEM "http://www.kinzan.net/dtd/kapp.dtd" >

<ktp:kapp name="simplegui" xmlns:ktp="http://www.kinzan.net">

    <ktp:domainList>
        <ktp:domain name="localhost" documentRoot="simplegui"/>
    </ktp:domainList>

    <!-- Widgets -->
    <ktp:widgetList>

        <ktp:widget name="Text">
            <ktp:attributeList>
                <ktp:local name="text" type="java.lang.String" defaultValue="No Text
Defined!"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"
locale="false">text.kwt</ktp:widgetTemplate>
        </ktp:widget>

        <ktp:widget name="ImageFromURL">

            <ktp:attributeList>
                <ktp:local name="URL" type="java.lang.String"
defaultValue="http://us.a1.yimg.com/us.yimg.com/i/my/hrt_1.gif"/>
                <ktp:local name="width" type="java.lang.String" defaultValue="60"/>
                <ktp:local name="height" type="java.lang.String" defaultValue="40"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"
locale="false">imageFromURL.kwt</ktp:widgetTemplate>

        </ktp:widget>
```

```
        <ktp:widget name="TextBox">

            <ktp:attributeList>
                <ktp:local name="name" type="java.lang.String" defaultValue="TextBox"/>
                <ktp:local name="isPassword" type="java.lang.String" defaultValue="N"/>
                <ktp:local name="size" type="java.lang.String"/>
                <ktp:local name="maxlength" type="java.lang.String"/>
                <ktp:local name="value" type="java.lang.String" defaultValue=""/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"
locale="false">GenericTextBox.kwt</ktp:widgetTemplate>

        </ktp:widget>

        <!-- Dropdown list form -->
        <ktp:widget name="DropDownList">

            <ktp:attributeList>
                <ktp:local name="prompt" type="java.lang.String"/>
                <ktp:local name="listPostName" type="java.lang.String"/>
                <ktp:local name="listSize" type="java.lang.String" defaultValue="1"/>
                <ktp:local name="options" type="java.lang.String[]"/>
                <ktp:local name="selected" type="java.lang.String"
defaultValue="!ThisNeedsToBeSet!"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"
locale="false">dropDownList.kwt</ktp:widgetTemplate>

        </ktp:widget>

        <ktp:widget name="SimpleList">

            <ktp:attributeList>
                <ktp:local name="listItems" type="java.lang.Vector"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"
locale="false">SimpleList.kwt</ktp:widgetTemplate>

        </ktp:widget>

        <ktp:widget name="SimpleHashTableView">

            <ktp:attributeList>
                <ktp:local name="hashTable" type="java.lang.HashTable"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"
locale="false">SimpleHashTable.kwt</ktp:widgetTemplate>

        </ktp:widget>

    </ktp:widgetList>

    <ktp:componentTypeList>

        <!-- The component name must be unique within this site -->
        <ktp:componentType name="Text" defaultMode="display">

            <ktp:description>This is a very simple text component</ktp:description>
```

```
            <ktp:attributeList>
                <ktp:reference name="text" widget="Text" attribute="text"/>
            </ktp:attributeList>

            <ktp:componentModeList>
                <ktp:componentMode name="display" widget="Text"/>
            </ktp:componentModeList>

        </ktp:componentType>

        <ktp:componentType name="URLImage" defaultMode="display">

            <ktp:description>This is a very simple gui component</ktp:description>

            <ktp:attributeList>
                <ktp:reference name="URL" widget="ImageFromURL" attribute="URL"/>
                <ktp:reference name="width" widget="ImageFromURL" attribute="width"/>
                <ktp:reference name="height" widget="ImageFromURL" attribute="height"/>
            </ktp:attributeList>

            <ktp:componentModeList>
                <ktp:componentMode name="display" widget="ImageFromURL"/>
            </ktp:componentModeList>

        </ktp:componentType>


        <ktp:componentType name="TextBox" defaultMode="display">
            <ktp:attributeList>
                <ktp:reference name="name" widget="TextBox" attribute="name"/>
                <ktp:reference name="isPassword" widget="TextBox"
attribute="isPassword"/>
                <ktp:reference name="size" widget="TextBox" attribute="size"/>
                <ktp:reference name="maxlength" widget="TextBox" attribute="maxlength"/>
                <ktp:reference name="value" widget="TextBox" attribute="value"/>
            </ktp:attributeList>

            <ktp:componentModeList>

                <ktp:componentMode name="display" widget="TextBox"/>

            </ktp:componentModeList>

        </ktp:componentType>

        <ktp:componentType name="DropDownList" defaultMode="display">

            <ktp:attributeList>
                <ktp:reference name="prompt" widget="DropDownList" attribute="prompt"/>
                <ktp:reference name="listPostName" widget="DropDownList"
attribute="listPostName"/>
                <ktp:reference name="listSize" widget="DropDownList"
attribute="listSize"/>
                <ktp:reference name="options" widget="DropDownList" attribute="options"/>
                <ktp:reference name="selected" widget="DropDownList"
attribute="selected"/>
            </ktp:attributeList>

            <ktp:componentModeList>

                <ktp:componentMode name="display" widget="DropDownList"/>

            </ktp:componentModeList>
```

```
        </ktp:componentType>

    </ktp:componentTypeList>
</ktp:kapp>

end of file: simplegui.kapp
```

2.GenericDataTable Component

1.13 Introduction

1.13.1 Audience

This document is intended for developers who will be working with the Kinzan Technology Platform. It explains how to use the GenericDataTable component.

1.13.2 Overview

This document describes the GenericDataTable component. This component takes a set of tabular data and displays in the desired format. The data is bound from an EJB, obtained through a business rule (session bean).

1.14 GenericDataTable Component

1.14.1 GenericDataTable Widget Definition

The GenericDataTable widget is defined in the file "genericdatatable.kapp". Here is the excerpt containing the widget definition.

```
1       <ktp:widget name="GenericDataTable">
2               <ktp:attributeList>
3                       <ktp:local name="content" type="java.util.Vector"/>
4                       <ktp:local name="header" type="java.util.Vector"/>
5                       <ktp:local name="footer" type="java.util.Vector"/>
6                       <ktp:local name="firstCol" type="java.util.Vector"/>
7                       <ktp:local name="lastCol" type="java.util.Vector"/>
8                       <ktp:local name="defaultStyleClass" type="java.lang.String"/>
9                       <ktp:local name="columnClasses" type="java.util.Vector"/>
10                      <ktp:local name="rowClasses" type="java.util.Vector"/>
11                      <ktp:local name="headerClasses" type="java.util.Vector"/>
12                      <ktp:local name="footerClasses" type="java.util.Vector"/>
13                      <ktp:local name="firstColClasses" type="java.util.Vector"/>
14                      <ktp:local name="lastColClasses" type="java.util.Vector"/>
15                      <ktp:local name="dataTypes" type="java.util.Vector"/>
16                      <ktp:local name="dataTypeClass" type="java.util.Vector"/>
17                      <ktp:local name="dataTypePrefix" type="java.util.Vector"/>
18                      <ktp:local name="dataTypeSuffix" type="java.util.Vector"/>
19                      <ktp:local name="dataTypeReplace" type="java.util.Vector"/>
20              </ktp:attributeList>
```

```
21              <ktp:widgetTemplate style="false"
                locale="false">GenericDataTable.kwt</ktp:widgetTemplate>
22    </ktp:widget>
```

Line 1 — The widget is defined with the name "GenericDataTable"

Lines 2-20 — The attributes of the widget are defined and given default values (more below).

Line 3 — The content of the table, represented as a vector of which each element is another vector (representing a row in the table). This is the only required attribute of the widget, it must have a value bound to it.

Lines 4-7 — The contents of the header row, footer row, and first/last columns.

Lines 8-14 — Vectors specifying style classes to be used in formatting the table and its rows/cells. The widget template will look for these style classes in the style file associated with whichever component implements it.

Line 15 — A vector of "datatypes". These are strings representing particular types of data, such as "StockSymbol" or "PriceIncrease". If the contents of "content" (line 3) make use of the GenericMetaData interface the GenericDataTable widget template can make use of this list of datatypes to associate the content of each cell of the table with a set of style classes and text changes (see 16-19 below).

Line 16 — A list of style classes corresponding to the contents of the "datatypes" list. These are the styles to use when displaying data of that type.

Line 17 — A list of prefixes corresponding to the contents of the "datatypes" list. A datatype's prefix will be prepended to the display content.

Line 18 — A list of suffixes corresponding to the contents of the "datatypes" list. A datatype's suffix will be appended to the display content.

Line 19 — A list of replacement text corresponding to the contents of the "datatypes" list. A datatype's replacement text will be displayed in place of the display content.

Lines 21 — The widget template. This points to the widget template file. There are no style or locale variants for this widget. However, there are device variants.

1.14.2 Device Variations

The GenericDataTable widget supports variants in html and WML. This is achieved by defining the appropriate variant templates for each of the widgets utilized in rendering the display for the component.

file: genericdatatable.kapp

```
<?xml version="1.0"?>
<!DOCTYPE ktp:kapp SYSTEM "http://www.kinzan.net/dtd/kapp.dtd" >

<ktp:kapp name="genericdatatable" xmlns:ktp="http://www.kinzan.net">

    <ktp:domainList>
        <ktp:domain name="localhost" documentRoot="genericdatatable"/>
    </ktp:domainList>

    <!-- Widgets -->
    <ktp:widgetList>
        <ktp:widget name="GenericDataTable">

            <ktp:attributeList>
                <ktp:local name="content" type="java.util.Vector"/>
                <ktp:local name="defaultStyleClass" type="java.lang.String"/>
                <ktp:local name="columnClasses" type="java.util.Vector"/>
                <ktp:local name="rowClasses" type="java.util.Vector"/>
                <ktp:local name="headerClasses" type="java.util.Vector"/>
                <ktp:local name="footerClasses" type="java.util.Vector"/>
                <ktp:local name="firstColClasses" type="java.util.Vector"/>
                <ktp:local name="lastColClasses" type="java.util.Vector"/>
                <ktp:local name="header" type="java.util.Vector"/>
                <ktp:local name="footer" type="java.util.Vector"/>
                <ktp:local name="firstCol" type="java.util.Vector"/>
                <ktp:local name="lastCol" type="java.util.Vector"/>
                <ktp:local name="dataTypes" type="java.util.Vector"/>
                <ktp:local name="dataTypeClass" type="java.util.Vector"/>
                <ktp:local name="dataTypePrefix" type="java.util.Vector"/>
                <ktp:local name="dataTypeSuffix" type="java.util.Vector"/>
                <ktp:local name="dataTypeReplace" type="java.util.Vector"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"
locale="false">GenericDataTable.kwt</ktp:widgetTemplate>
        </ktp:widget>
    </ktp:widgetList>

    <ktp:componentTypeList>
        <!-- The component name must be unique within this site -->
        <ktp:componentType name="GenericDataTable" defaultMode="display">
            <ktp:attributeList>
                <ktp:reference name="content"
                               widget="GenericDataTable"
                               attribute="content"/>

                <ktp:reference name="defaultStyleClass"
                               widget="GenericDataTable"
                               attribute="defaultStyleClass"/>

                <ktp:reference name="columnClasses"
                               widget="GenericDataTable"
                               attribute="columnClasses"/>

                <ktp:reference name="rowClasses"
                               widget="GenericDataTable"
                               attribute="rowClasses"/>

                <ktp:reference name="headerClasses"
```

```
                        widget="GenericDataTable"
                        attribute="headerClasses"/>

                <ktp:reference name="footerClasses"
                        widget="GenericDataTable"
                        attribute="footerClasses"/>

                <ktp:reference name="firstColClasses"
                        widget="GenericDataTable"
                        attribute="firstColClasses"/>

                <ktp:reference name="lastColClasses"
                        widget="GenericDataTable"
                        attribute="lastColClasses"/>

                <ktp:reference name="header"
                        widget="GenericDataTable"
                        attribute="header"/>

                <ktp:reference name="footer"
                        widget="GenericDataTable"
                        attribute="footer"/>

                <ktp:reference name="firstCol"
                        widget="GenericDataTable"
                        attribute="firstCol"/>

                <ktp:reference name="lastCol"
                        widget="GenericDataTable"
                        attribute="lastCol"/>

                <ktp:reference name="dataTypes"
                        widget="GenericDataTable"
                        attribute="dataTypes"/>

                <ktp:reference name="dataTypeClass"
                        widget="GenericDataTable"
                        attribute="dataTypeClass"/>

                <ktp:reference name="dataTypePrefix"
                        widget="GenericDataTable"
                        attribute="dataTypePrefix"/>

                <ktp:reference name="dataTypeSuffix"
                        widget="GenericDataTable"
                        attribute="dataTypeSuffix"/>

                <ktp:reference name="dataTypeReplace"
                        widget="GenericDataTable"
                        attribute="dataTypeReplace"/>

            </ktp:attributeList>
            <ktp:componentModeList>
                <ktp:componentMode name="display" widget="GenericDataTable"/>
            </ktp:componentModeList>
        </ktp:componentType>
    </ktp:componentTypeList>
</ktp:kapp>

end of file: genericTable.kapp
```

3.Weather Component

1.15 Introduction

1.15.1 Audience

This document is intended for developers who will be working with the Kinzan Technology Platform. It explains some aspects of the KTP 4 component architecture by walking through an example component.

1.15.2 Overview

This document describes a simple Kinzan component--the weather component. The weather component looks up weather information based on an input ZIP code. The component is a simple component in that it has no extra classes in order to make it work. All of its functionality is described in the KApp, widget template and state diagram files.

1.16 Weather Component

The weather component is defined in a KApp file named weather.kapp. We will walk through the important elements of weather.kapp and see what they do.

1.16.1 Weather Widget Definition

The first important piece of the weather KApp file is the widget definition. A widget is a view. It is completely independent from the data it displays. It may have multiple representations based on device and locale.

```
1     <ktp:widget name="WeatherDisplay">
2
3       <ktp:attributeList>
4          <ktp:local name="title" type="java.lang.String" defaultValue="Weather"/>
5          <ktp:local name="prompt" type="java.lang.String" defaultValue="Enter Zip:"/>
6          <ktp:local name="submitValue1" type="java.lang.String" defaultValue="Go"/>
7          <ktp:local name="submitValue2" type="java.lang.String"/>
8       </ktp:attributeList>

9       <ktp:widgetAssetList>
10        <ktp:widgetAsset name="Go" asset="GoButton" overwriteable="true"/>
11      </ktp:widgetAssetList>

12      <ktp:widgetTemplate style="false"
locale="false">weatherDisplayTemplate.kwt</ktp:widgetTemplate>
```

```
13    </ktp:widget>
```

Line 1 — The widget is defined with the name "WeatherDisplay"

Lines 3-8 — The attributes of the widget are defined and given default values. The attributes are defined in order that a component definition can bind particular values to them. Each of these attribute names corresponds to a template variable with the name "widget.<attributeName>". The widget template, "weatherDisplayTemplate.kwt", is included at the end of this document. Look at that and compare its contents with this segment of the KApp file.

Lines 9-11 — The widget asset list. This defines assets that are used by the widget, such as a button. Line 9 defines a Go button asset that is overwriteable. The `overwriteable` attribute specifies whether or not the asset can be changed by derived widgets.

Line 12 — The widget template. This points to the widget template file. There are no style or locale variants for this widget.

1.16.2 Weather Component Definition

The weather widget "WeatherDisplay" is the only view object that the Weather Component needs in order to display itself. It is now time, therefore, to define the component. The component has two modes--one that displays weather detail information and one that displays a weather forecast. Each defines the widget it uses (which is WeatherDisplay in both cases), and the bindings to that widget.

```
1     <ktp:componentType name="Weather" defaultMode="display">

2       <ktp:description>This is a very simple weather widget.</ktp:description>

3         <ktp:attributeList>
4           <ktp:reference name="title" widget="WeatherDisplay" attribute="title"/>
5         </ktp:attributeList>

6         <ktp:componentList>
7           <ktp:component name="weatherImage" type="URLImage"/>
8           <ktp:component name="zipInput" type="TextBox"/>
9           <ktp:component reference="myLogin"/>
10      </ktp:componentList>
```

Line 1 — The component type "Weather" is defined. It has a default mode named "display".

Line 2 — The component description. This does not affect the rendering system.

Lines 3-5 — The component has one attribute, `title`, which is bound to the value of `title` in the WeatherDisplay widget.

Lines 6-10 Lists the child components used by this component. The `weatherImage` and `zipInput` components will be instantiated for this component instance. The `myLogin` component is defined in another location (login.kapp).

Now that the component attributes and child components are defined, the component modes are defined. Each mode of a component has one widget. The widget is topmost display component. Each mode then has a set of mappings, which bind child components into zones, and a set of bindings, which map data to widget or component attributes.

```
11      <ktp:componentModeList>

12        <ktp:componentMode name="display" widget="WeatherDisplay">
13          <ktp:zone name="weatherInfo" component="weatherImage"/>
14          <ktp:zone name="inputArea" component="zipInput"/>
15          <ktp:zone name="loginZone" component="myLogin"/>
16          <ktp:bindings>

17            <ktp:binding component="weatherImage" attribute="URL"
defaultValue="http://209.70.191.111/zipcode/9/90210.gif">
18                <ktp:from userContext="url"/>
19            </ktp:binding>

20            <ktp:binding component="weatherImage" attribute="width">
21              <ktp:from literal="150"/>
22            </ktp:binding>

23            <ktp:binding component="weatherImage" attribute="height">
24              <ktp:from literal="50"/>
25            </ktp:binding>

26            <ktp:binding component="zipInput" attribute="value" defaultValue="90210">
27              <ktp:from userContext="zip"/>
28            </ktp:binding>
              ...
29            <ktp:binding widgetAttribute="prompt">
30              <ktp:from bundle="com.kinzan.example.weather.Weather_res"
attribute="ENTER_ZIP"/>
31            </ktp:binding>

32            <ktp:binding attribute="title">
33              <ktp:from bundle="com.kinzan.example.weather.Weather_res"
attribute="TITLE"/>
34            </ktp:binding>
35          </ktp:bindings>
36      </ktp:componentMode>
```

Line 12 A component mode is defined. The mode is named "display" and it uses the widget "WeatherDisplay". The mode name is referenced in the component's state diagram. This will be shown later.

Lines 13-15 The zones of the WeatherDisplay widget are mapped to the child components of the Weather Component.

Lines 16-35 The attributes of the child components and widgets are bound to values. They can be bound to literals, user context attributes, resources bundle values, and other things.

Lines 17-19 The `URL` attribute of the `weatherImage` child component is bound to the user context attribute "URL". Line 17 also shows a default value for the URL attribute--this will be used before the binding can be run.

Lines 20-25 The `width` and `height` attributes of the `weatherImage` component are bound to literal values "150" and "50".

Lines 26-28 The `value` attribute of the `zipInput` component is given the defaultValue "90210" and is bound to the `userContext` attribute named "zip".

Lines 29-31 The `prompt` attribute of the component's display widget (WeatherDisplay) is bound to the value found in the resource bundle `com.kinzan.example.weather.Weather_res` using the key `ENTER_ZIP`.

Lines 32-34 The `title` attribute of the component is bound to the value found in the resource bundle `com.kinzan.example.weather.Weather_res` using the key `TITLE`.

So far we have defined the first mode of the component. The component has a second mode, "forecastDisplay", that displays a five-day weather forecast.

```
37      <ktp:componentMode name="forecastDisplay" widget="WeatherDisplay">
38          <ktp:zone name="weatherInfo" component="weatherImage"/>
39          <ktp:zone name="inputArea" component="zipInput"/>
40          <ktp:zone name="loginZone" component="myLogin"/>
41          <ktp:bindings>

42            <ktp:binding component="weatherImage" attribute="URL"
defaultValue="http://209.70.191.111/forecast/9/90210.gif">
43              <ktp:from userContext="url"/>
44            </ktp:binding>
              ...
45          </ktp:bindings>
46        </ktp:componentMode>

47      </ktp:componentModeList>
```

Line 37 The `forecastDisplay` mode of the component is defined. It also uses the WeatherDisplay widget.

Lines 38-40 Since this mode uses the same widget as the `display` mode, it will need to define components for each of the same zones. Since we want to display the same components, these zone definitions are the same as the display mode's zone definitions. Note that the zones for each component mode do not need to match, but we have done this because we have a very simple component.

Lines 41-45 Bindings are repeated for every component mode. This is because the component mode may use entirely different widgets and components, and it may need entirely different bindings.

Now that the modes of our component are defined, we will want to define how the component switches from one mode to another. This is done by associating a state diagram with this component.

```
48      <ktp:stateDiagram name="Weather"/>

49    </ktp:componentType>
```

Line 48 The stateDiagram named "Weather" is associated with this component. When the state manager receives requests generated by an instance of this component, those requests will first be evaluated against the Weather state diagram.

1.16.3 Weather Component State Diagram

Each component may have one state diagram associated with it. If a component does not have a state diagram, the component does not handle any events.

The weather component uses a state diagram named "Weather". The Weather state diagram is defined in the state diagram file "Weather.ksd".

The new state diagram tag <ktp:componentState> is the component equivalent of a displayState. When a component ends up in a componentState, the state manager sets its mode to the name of the componentState. In this way, a componentState has a one-to-one relationship to component modes defined in the KApp file.

```
1     <?xml version="1.0"?>

2     <!DOCTYPE ktp:stateDiagram SYSTEM "http://www.kinzan.net/dtd/ksd.dtd">

3     <ktp:stateDiagram name="Weather" startState="display"
xmlns:ktp="http://www.kinzan.net">

       <!--if an action throws an exception it may be caught by a <ktp:catch> statement-->
4     <ktp:catch exception="net.kinzan.state.action.InvalidParameterException"
5                 state="display"/>

6     <ktp:componentState name="display">
7       <ktp:transition event="Forecast" state="xSetForecast"/>
8       <ktp:transition event="Go" state="xSetDetail"/>
9     </ktp:componentState>

10    <ktp:componentState name="forecastDisplay" permission="forecast" >
11      <ktp:transition event="Detail" state="xSetDetail"/>
```

```
12      <ktp:transition event="Go" state="xSetForecast"/>
13    </ktp:componentState>
```

Lines 1-2 XML file headers define the file type.

Lines 4-5 A catch statement. Rather than coding to the FAILURE event (of KTP 3), the actual exception thrown by an Action can be caught. The catch is similar to a state transition since it defines a result state when the exception is caught.

Lines 6-9 The component mode "display" is represented in the state diagram. From the display mode, two transitions may occur. The "Forecast" event will trigger a transition into the "xSetForecast" Action (described later). The "Go" event will trigger a transition into the "xSetDetail" Action (also described later).

Lines 10-13 The component mode "forecastDisplay" is represented in the state diagram. It has two transitions.

The Actions (formerly PipelineComponents) required for the Weather Component are simple and were therefore coded as inline JavaScript.

```
14    <ktp:actionState name="xSetForecast" permission="forecast" >
15      <ktp:script language="JavaScript">
16        zipcode = actionEvent.getAttribute( "zip" );
17        userContext.setAttribute( "zip", zipcode );
18        userContext.setAttribute( "url", "http://209.70.191.111/forecast/"
19                    + zipcode.substring( 0, 1 ) + "/" + zipcode + ".gif" );
20        actionEvent.setName( "SetForecast" );
21      </ktp:script>
22      <ktp:transition event="SetForecast" state="forecastDisplay"/>
23    </ktp:actionState>

24    <ktp:actionState name="xSetDetail">
25      <ktp:script language="JavaScript">
26       zipcode = actionEvent.getAttribute( "zip" );
27        userContext.setAttribute( "zip", zipcode );
28        userContext.setAttribute( "url", "http://209.70.191.111/zipcode/"
29                    + zipcode.substring( 0, 1 ) + "/" + zipcode + ".gif" );
30        actionEvent.setName( "SetDetail" );
31      </ktp:script>
32      <ktp:transition event="SetDetail" state="display"/>
33    </ktp:actionState>
```

Line 14 Defines an ActionState named "xSetForecast". The State requires the permission named "forecast" in order to be executed.

Lines 15-21 Defines an inline JavaScript definition for this component. This is the server side code that will be executed when this state is entered. Alternatively an Action class (a Java class that implements the `net.kinzan.state.action.Action` interface) could be defined for the ActionState.

Line 22 A transition for the "SetForecast" event is defined. The ActionState returns this event when it has set the forecast information.

Lines 23-33 The xSetDetail ActionState is defined in much the same way as the xSetForecast was.

file: weather.kapp

```
<?xml version="1.0"?>
<!DOCTYPE ktp:kapp SYSTEM "http://www.kinzan.net/dtd/kapp.dtd" >

<ktp:kapp name="weather" xmlns:ktp="http://www.kinzan.net">

  <!-- Define domains -->
  <ktp:domainList>
    <ktp:domain name="localhost" documentRoot="weather"/>
  </ktp:domainList>

  <!-- Includes -->
  <ktp:includeList>
    <ktp:include>simplegui.kapp</ktp:include>
    <ktp:include>assetlib.kapp</ktp:include>
    <ktp:include>login.kapp</ktp:include>
  </ktp:includeList>

  <!-- Widgets -->
  <ktp:widgetList>

    <ktp:widget name="WeatherDisplay">

      <ktp:attributeList>
        <ktp:local name="title" type="java.lang.String" defaultValue="Weather"/>
        <ktp:local name="prompt" type="java.lang.String" defaultValue="Enter Zip:"/>
        <ktp:local name="submitValue1" type="java.lang.String" defaultValue="Go"/>
        <ktp:local name="submitValue2" type="java.lang.String"/>
      </ktp:attributeList>

      <ktp:widgetAssetList>
        <ktp:widgetAsset name="Go" asset="GoButton" overwriteable="true"/>
      </ktp:widgetAssetList>

      <ktp:widgetTemplate style="false"
locale="false">weatherDisplayTemplate.kwt</ktp:widgetTemplate>

    </ktp:widget>

  </ktp:widgetList>

  <ktp:componentTypeList>

    <!-- The component name must be unique within this site -->
    <ktp:componentType name="Weather" defaultMode="display">

      <ktp:description>This is a very simple weather widget</ktp:description>

      <ktp:attributeList>
        <ktp:reference name="title" widget="WeatherDisplay" attribute="title"/>
      </ktp:attributeList>

      <ktp:componentList>
        <ktp:component name="weatherImage" type="URLImage"/>
        <ktp:component name="zipInput" type="TextBox"/>
        <ktp:component reference="myLogin"/>
      </ktp:componentList>
```

```
      <ktp:componentModeList>

<ktp:componentMode name="display" widget="WeatherDisplay">
          <ktp:zone name="weatherInfo" component="weatherImage"/>
          <ktp:zone name="inputArea" component="zipInput"/>
          <ktp:zone name="loginZone" component="myLogin"/>
          <ktp:bindings>

            <ktp:binding component="weatherImage" attribute="URL"
defaultValue="http://209.70.191.111/zipcode/9/90210.gif">
              <ktp:from userContext="url"/>
            </ktp:binding>

            <ktp:binding component="weatherImage" attribute="width">
              <ktp:from literal="150"/>
            </ktp:binding>

            <ktp:binding component="weatherImage" attribute="height">
              <ktp:from literal="50"/>
            </ktp:binding>

            <ktp:binding component="zipInput" attribute="value" defaultValue="90210">
              <ktp:from userContext="zip"/>
            </ktp:binding>


            <ktp:binding component="zipInput" attribute="size">
              <ktp:from literal="7"/>
            </ktp:binding>

            <ktp:binding component="zipInput" attribute="name">
              <ktp:from literal="zip"/>
            </ktp:binding>

            <ktp:binding widgetAttribute="submitValue2">
              <ktp:from literal="Forecast"/>
            </ktp:binding>

            <ktp:binding widgetAttribute="prompt">
              <ktp:from bundle="com.kinzan.example.weather.Weather_res"
attribute="ENTER_ZIP"/>
            </ktp:binding>

            <ktp:binding attribute="title">
              <ktp:from bundle="com.kinzan.example.weather.Weather_res"
attribute="TITLE"/>
            </ktp:binding>
          </ktp:bindings>
        </ktp:componentMode>

        <ktp:componentMode name="forecastDisplay" widget="WeatherDisplay">
          <ktp:zone name="weatherInfo" component="weatherImage"/>
          <ktp:zone name="inputArea" component="zipInput"/>
          <ktp:zone name="loginZone" component="myLogin"/>
          <ktp:bindings>

            <ktp:binding component="weatherImage" attribute="URL"
defaultValue="http://209.70.191.111/forecast/9/90210.gif">
              <ktp:from userContext="url"/>
            </ktp:binding>

            <ktp:binding component="weatherImage" attribute="width">
              <ktp:from literal="396"/>
```

```
            </ktp:binding>

            <ktp:binding component="weatherImage" attribute="height">
              <ktp:from literal="176"/>
            </ktp:binding>

            <ktp:binding component="zipInput" attribute="value" defaultValue="90210">
              <ktp:from userContext="zip"/>
            </ktp:binding>

            <ktp:binding component="zipInput" attribute="name">
              <ktp:from literal="zip"/>
            </ktp:binding>

            <ktp:binding component="zipInput" attribute="size">
              <ktp:from literal="7"/>
            </ktp:binding>

            <ktp:binding widgetAttribute="submitValue2">
              <ktp:from literal="Detail"/>
            </ktp:binding>

            <ktp:binding widgetAttribute="prompt">
              <ktp:from bundle="com.kinzan.example.weather.Weather_res"
attribute="ENTER_ZIP"/>
            </ktp:binding>

            <ktp:binding attribute="title">
              <ktp:from bundle="com.kinzan.example.weather.Weather_res"
attribute="TITLE"/>
            </ktp:binding>
          </ktp:bindings>
        </ktp:componentMode>

      </ktp:componentModeList>

      <ktp:stateDiagram name="Weather"/>

    </ktp:componentType>

  </ktp:componentTypeList>


  <ktp:KPDList>
    <ktp:KPD name="firstComponent" title="FirstComponent">
      <ktp:description>My first component</ktp:description>
      <ktp:component name="weatherComponent" type="Weather"
uniqueName="examples.weather"/>
    </ktp:KPD>
  </ktp:KPDList>


</ktp:kapp>

end of file: weather.kapp
```

file: weatherDisplayTemplate.kwt

```
<table>
    <tr>
        <td>
            <ktp:text text="widget.title"/>
        </td>
    </tr>
    <tr>
        <td>
            <ktp:zone name="weatherInfo"/>
        </td>
    </tr>

    <tr>
        <td>
            <ktp:form>
                <ktp:text text="widget.prompt"/>
                <ktp:zone name="inputArea"/>
                <ktp:submit value="widget.submitValue1" image="widget.Go"/>
                <ktp:submit value="widget.submitValue2"/>
            </ktp:form>
        </td>
    </tr>

</table>
<table>
    <tr>
        <td>
            <ktp:zone name="loginZone"/>
        </td>
    </tr>
</table>
```

end of file: weatherDisplayTemplate.kwt

1.17 KApp requirements to support application security

The weather component has a protected mode. To allow access to application users we will need to map the component instance with the component permission required to access it. This requirement means we have to give the component a site-wide unique name using the uniqueName attribute.

```
<ktp:kapp name="weather" xmlns:ktp="http://www.kinzan.net">

-
    3 <ktp:KPDList>
    4     <ktp:KPD name="firstComponent" title="FirstComponent">
    5        <ktp:description>My first component</ktp:description>
    6        <ktp:component name="weatherComponent" type="Weather" uniqueName="examples.weather"/>
    7     </ktp:KPD>
    8 </ktp:KPDList>
-
```

```
</ktp:kapp>
```

Line 6 the uniqueName attribute is added to the component instance definition so that it can be bound to security mappings in the KSec file.

1.18 ComponentMode ACLs

The weather application requires a user to have permission to get the forecast. After all, not just anybody can know tomorrow's weather. To enforce this requirement we must add permission to the forecast mode. To be quite secure, we will protect both the forecast actionState that transitions to the forecast displayState and the displayState itself.

This is to safeguard future changes where a new transition may be added to the displayState bypassing the actionState.

```
<ktp:stateDiagram name="Weather" startState="display" xmlns:ktp="http://www.kinzan.net">

...
9     <ktp:componentState name="forecastDisplay" permission="forecast" >
10        <ktp:transition event="Detail" state="xSetDetail"/>
11        <ktp:transition event="Go" state="xSetForecast"/>
12    </ktp:componentState>
13    <ktp:actionState name="xSetForecast" permission="forecast" >
14    <ktp:script language="JavaScript">
          zipcode = actionEvent.getAttribute( "zip" );
          userContext.setAttribute( "zip", zipcode );
          userContext.setAttribute( "url", "http://209.70.191.111/forecast/"
          + zipcode.substring( 0, 1 ) + "/" + zipcode + ".gif" );
          actionEvent.setName( "SetForecast" );
15    </ktp:script>
          <ktp:transition event="SetForecast" state="forecastDisplay"/>
16    </ktp:actionState>


</ktp:stateDiagram>
```

Line 9 Permission attribute is added to forecast displayState – the attribute value is the action name of the permission that will be checked at runtime.

Line 13 Permission attribute is added to forecast actionState

1.19 Dealing with Security Exceptions

The Kinzan platform deals with security transitions through exceptions.

There are 3 main exceptions to consider in your application KSD file:

- javax.security.auth.login.FailedLoginException

User is either not logged in or cannot be logged in (bad username/password)

- net.kinzan.security.auth.NoAuthorityException

  User is logged in but lacks necessary permissions – 2 possible ways to handle this is to offer the user a chance to logout and re-login as a user with correct privileges or to politely inform the user they have attempted an action they are not authorized to make.

- javax.security.auth.login.LoginException

  Parent exception to all login exceptions including the previous 2 exceptions

  To forward a user to a security login page use ksd LoginWizard state LoginDisplay and they will be forwarded to a Login KPD with the login component as the root component giving the user a chance to login to the application. Remember that the StateManager goes from top to bottom in matching an exception to transition. So be sure to have the first 2 exceptions listed above listed before the 3rd one.

```
<ktp:stateDiagram name="Application" startState="one" xmlns:ktp="http://www.kinzan.net">

1     <!-- User Failed to Login -->
2       <ktp:catch exception="javax.security.auth.login.FailedLoginException" state="
LoginDisplay" wizard="LoginWizard"/>

3     <!--User has successfully logged in but lacks necessary permissions to access state
-->
4     <ktp:catch exception="net.kinzan.security.auth.NoAuthorityException" state="NoAuthority"/>

5     <!-- General Exception for all JAAS login security situations -->
6     <ktp:catch exception="javax.security.auth.login.LoginException" state="NoAuthority"/>

7     <ktp:catch exception="java.lang.Exception" state="one"/>

8     <ktp:displayState name="one" kpd="firstComponent"/>

9     <ktp:displayState name="NoAuthority" forward="/errors/NoAuthority.html"/>

</ktp:stateDiagram>
```

Line 2 Catch the bad password and redisplay the LoginWizard LoginDisplay state

Line 4 Catch the NoAuthority exception and move to NoAuthority state which will just print out an HTML file with an explaination of the error.

Line 6 Catch the general LoginException and send to an error page.

1.20 Application Security (KSec)

For the weather example, we need to supply a Kinzan Security (KSec) file to map component permissions to application roles. We have two significant roles to point out. "ADMIN" is any application admin user that will be granted any

permission required by weather component, whereas the "USER" application role will only be granted "forecast" permission.

```
1   <ktp:ksec name="weather" xmlns:ktp="http://www.kinzan.net">

2 <!--
3     Application Roles
4     List all roles available in this site to users
5     Assign to each role the component access permissions granted to the role
6 -->
7     <ktp:appRole>
8         <ktp:role name="SUPER">
9             <!-- allaccess grants all necessary permissions to use everything -->
10              <!-- convenient shorthand -->
11              <ktp:componentAccess allaccess="true"/>
12          </ktp:role>
13          <ktp:role name="ADMIN">
14              <!-- component permissions can either be spelled out or set attribute
15              allpermission to true this is an example of the allpermission case
16              -->
17              <ktp:componentPermission componentUniqueName="examples.weather" allpermission="true"
18          </ktp:role>
19          <ktp:role name="USER">
20              <!-- component permissions can either be spelled out or set attribute allpermission
21              to true this is an example of listing permissions
22              -->
23              <ktp:componentPermission componentUniqueName="examples.weather" >
24                  <ktp:permission name="forecast"/>
25              </ktp:componentPermission>
26          </ktp:role>
27          <!-- Default to All Users -->
28          <ktp:role>
29          </ktp:role>
30      </ktp:appRole>

31  </ktp:ksec>
```

Line 11 is a site wide super permission that encompasses all permissions necessary for access – equivalent of root on UNIX and administrator on NT but only as far as the application is concerned.

Line 17 grant all permissions for a specific component instance to the encapsulating role in this case it is the "ADMIN" role

Lines 23-25 A specific named permission on a component instance is granted. In this case, forecast permission is granted to "USER" role.

file: Weather.ksd

```
<?xml version="1.0"?>

<!DOCTYPE ktp:stateDiagram SYSTEM "http://www.kinzan.net/dtd/ksd.dtd">

<ktp:stateDiagram name="Weather" startState="display" xmlns:ktp="http://www.kinzan.net">

<!-- if an action throws an exception it may be caught by a <ktp:catch> statement -->
<ktp:catch exception="net.kinzan.state.action.InvalidParameterException"
           state="display"/>

<ktp:componentState name="display">
    <ktp:transition event="Forecast" state="xSetForecast"/>
    <ktp:transition event="Go" state="xSetDetail"/>
  </ktp:componentState>

  <ktp:componentState name="forecastDisplay" permission="forecast" >
    <ktp:transition event="Detail" state="xSetDetail"/>
    <ktp:transition event="Go" state="xSetForecast"/>
  </ktp:componentState>

  <ktp:actionState name="xSetForecast" permission="forecast" >
    <ktp:script language="JavaScript">
      zipcode = actionEvent.getAttribute( "zip" );
      userContext.setAttribute( "zip", zipcode );
      userContext.setAttribute( "url", "http://209.70.191.111/forecast/" +
zipcode.substring( 0, 1 ) + "/" + zipcode + ".gif" );

      actionEvent.setName( "SetForecast" );
    </ktp:script>
    <ktp:transition event="SetForecast" state="forecastDisplay"/>
  </ktp:actionState>

  <ktp:actionState name="xSetDetail">
    <ktp:script language="JavaScript">
      zipcode = actionEvent.getAttribute( "zip" );
      userContext.setAttribute( "zip", zipcode );
      userContext.setAttribute( "url", "http://209.70.191.111/zipcode/" +
zipcode.substring( 0, 1 ) + "/" + zipcode + ".gif" );

      actionEvent.setName( "SetDetail" );
    </ktp:script>
    <ktp:transition event="SetDetail" state="display"/>
  </ktp:actionState>

</ktp:stateDiagram>
```

end of file: Weather.ksd

4.Stock Quote Component

1.21 Introduction

1.21.1 Audience

This document is intended for developers who will be working with the Kinzan Technology Platform. It explains some aspects of the KTP 4 component architecture by walking through an example component. This document assumes that the reader has reviewed the weather component document (above) and understands the terms described therein.

1.21.2 Overview

This document describes a sample Kinzan component--the stockquote component. The stockquote component gives a listing of current stock quotes in a generic table widget. The stock quotes are bound from an EJB, obtained through a business rule (session bean), an aspect of the platform not utilized by the weather component.

1.22 Stock Quote Component

The stockquote component is defined in a KApp file named stockquote.kapp. We will walk through the important elements of stockquote.kapp and see what they do.

1.22.1 Stockquote Widget Definition

The first important piece of the stockquote KApp file is the widget definition. A widget is a view. It is completely independent from the data it displays. It may have multiple representations based on device and locale.

```
1       <ktp:widget name="StockDisplay">

2            <ktp:attributeList>
3                 <ktp:local name="prompt" type="java.lang.String"/>
4            </ktp:attributeList>
5
6            <ktp:widgetTemplate style="false"
                                 locale="false">
7                StockDisplayTemplate.kwt
8            </ktp:widgetTemplate>
```

```
9       </ktp:widget>
```

Line 1 The widget is defined with the name "StockDisplay"

Lines 2-4 The attributes of the widget are defined and given default values. The attributes are defined in order that a component definition can bind particular values to them. Each of these attribute names corresponds to a template variable with the name "widget.<attributeName>". The widget template, "stockDisplayTemplate.kwt", is included at the end of this document. Look at that and compare its contents with this segment of the kapp file.

Lines 6-8 The widget template. This points to the widget template file. There are no style or locale variants for this widget. However, there are device variants.

The stockquote component makes use of a generic table widget to display its stock data. This widget and its associated generic component are defined in a KApp file included into the stockquote.kapp file. Please refer to the explanation of the GenericDataTable Component for details of its associated attributes.

1.22.2 Stock Quote Component Definition

As with the weather component, the stock quote component utilizes one main widget to present its view to the user. The widget "StockDisplay" contains the templates for each device supported by this component. Adding a device variation only requires adding the appropriate device variant templates to the widgets involved. The component itself is separated from the view and does not need to be aware of the actual view rendered. The StockDisplay widget includes zones that are mapped to other components containing other presentation elements. The stockquote component only contains one mode, which uses the StockDisplay widget.

```
1       <ktp:componentType name="StockQuote" defaultMode="display">

2           <ktp:description>This is a stock quote component</ktp:description>

3           <ktp:attributeList>
4               <ktp:local name="title"
                           type="java.lang.String"
                           defaultValue="Stocks"/>

5               <ktp:local name="defaultSymbols"
                           type="java.lang.String"
                           defaultValue="^DJI,^IXIC,^XAX"/>

6               <ktp:local name="dataType"
                           type="java.lang.String"
                           defaultValue="symbol,price,change,date"/>
```

```
7           </ktp:attributeList>


8           <ktp:componentList>
9               <ktp:component name="symbolInput" type="TextBox"/>
10              <ktp:component name="dataDisplay"
                               type="GenericDataTable"
                               style="stocktable"/>
11          </ktp:componentList>
```

Line 1 The component type "stockquote" is defined. It has a default mode named "display".

Line 2 The component description. This does not affect the rendering system.

Lines 3-7 The component has three attributes. Each attribute is local to the component and contain default values.

Lines 8-11 Lists the child components used by this component. Each child component is local to this component. The "GenericDataTable" component is the generic table component while the "TextBox" component is the same component type used by the weather component for a simple text input field.

Differing from the weather component, the stockquote component has another section in its definition to list the model beans that will be used by this component to obtain the stock quote data.

```
12          <ktp:modelBeanList>

13              <ktp:modelBean name="StockSymbols">
14                  <ktp:rule name="StockTool" method="getStockSymbols"/>
15                  <ktp:param name="param1"
                        type="java.lang.String"
                        defaultValue="guest">
16                      <ktp:from userContext="accountID"/>
17                  </ktp:param>

18                  <ktp:param name="param2" type="java.lang.Vector">
19                      <ktp:from attribute="defaultSymbols"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
20                  </ktp:param>

21                  <ktp:param name="param3" type="java.lang.String"  defaultValue="SUNW">
22                      <ktp:from userContext="net.kinzan.stockquote.currentSymbol"/>
23                  </ktp:param>
24              </ktp:modelBean>

25              <ktp:modelBean name="StockData">
26                  <ktp:rule name="StockQuote" method="getStockQuotes"/>
27                  <ktp:param name="param1" type="java.util.Vector">
28                      <ktp:from modelBean="StockSymbols"/>
29                  </ktp:param>
30                  <ktp:param name="param2" type="java.util.Vector">
31                      <ktp:from attribute="dataType"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
```

```
32                </ktp:param>
33              </ktp:modelBean>

34          </ktp:modelBeanList>
```

Line 13 A model bean is defined by the name "StockSymbols". This name is the name that the rest of the component definition can refer to this model bean by.

Lines 14 A model bean is returned by calling a specific method in a business rule, as defined in this line. The business rule needs to be registered with a bean manager in the system. The bean managers that this KApp requires are listed at the top of the KApp file.

Lines 15-23 These lines define the parameters to pass to the business rule method defined in line 14 in order to generate the required model bean correctly. The parameters must be listed in the order that the interface to the business rule method is defined.

Line 19 This parameter definition is of note because it contains a transformation attribute, which defines a class that the component service will utilize to transform the input type of the "defaultSymbols" attribute of the component to a Vector, which is the type that the business rule expects for this parameter.

Lines 24-33 These lines define another model used by the stockquote component, "StockData". Note that as one parameter to the business rule, the entire "StockSymbols" model bean is passed. This dependency is resolved by the component service, in that it only instantiates model beans as needed. This will work as long as there are no cyclic references.

There is one more item worth mentioning concerning the model beans. In our implementation, the "StockSymbols" bean and "StockTool" business rule are deployed remotely and managed by the RemoteEJBBeanManager, while the "StockData" bean and "StockQuote" business rule are deployed locally in the platform's build in EJB container and managed by the LocalEJBBeanManager. Despite this, their definition is the same, and the component utilizes them in the same way.

Now that we have defined the model beans, child components, and attributes, we are ready to define the stock quote component's one mode, "display". Since we have reviewed bindings in the weather component, we will only highlight the different details of the stock quote's mode definition.

```
34          <ktp:componentModeList>

35              <ktp:componentMode name="display" widget="StockDisplay">
36                  <ktp:zone name="inputArea" component="symbolInput"/>
```

```
37                    <ktp:zone name="displayArea" component="dataDisplay"/>
38                    <ktp:bindings>

                        .
                        .

39                        <ktp:binding component="dataDisplay" attribute="content">
40                            <ktp:from modelBean="StockData"/>
41                        </ktp:binding>
42                        <ktp:binding component="dataDisplay" attribute="header">
43                            <ktp:from literal="Symbol,Price,Change,Date"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
44                        </ktp:binding>

                        .
                        .

45                    </ktp:bindings>

46                </ktp:componentMode>

47            </ktp:componentModeList>

48            <ktp:stateDiagram name="Stockquote"/>

49        </ktp:componentType>
```

Line 35 The `display` mode of the component is defined. It uses the StockDisplay widget.

Lines 36-37 The zone mappings for this component.

Lines 39-40 The binding to the `content` attribute of the `dataDisplay` component is the `StockData` model bean.

Lines 42-44 The binding to the `header` attribute of the `dataDisplay` component is from a string literal. However, the attribute is a Vector, so a transformation is applied.

Line 48 The state diagram for the stock quote component is defined here.

1.22.3 Stock Quote Component State Diagram

Like the weather component, the stock quote component utilizes a state diagram to define the application flow and business logic. The stock quote state diagram is fairly simple, since it needs only to set the current requested symbols in the user context and allow the component bindings and model beans to retrieve the required quotes and display them through the widgets.

```
1    <ktp:stateDiagram name="Stockquote" startState="display"
                       xmlns:ktp="http://www.kinzan.net">

2        <ktp:catch exception="java.lang.Exception" state="display"/>
```

```
3        <ktp:componentState name="display">
4            <ktp:transition event="Go" state="xSetSymbol"/>
5        </ktp:componentState>


6        <ktp:actionState name="xSetSymbol">
7            <ktp:script language="JavaScript">
8              symbol = actionEvent.getAttribute( "symbol" );
9              userContext.setAttribute( "net.kinzan.stockquote.currentSymbol", symbol );

10             actionEvent.setName( "SetSymbol" );
11           </ktp:script>
12           <ktp:transition event="SetSymbol" state="display"/>
13       </ktp:actionState>

14   </ktp:stateDiagram>
```

Line 1 — The state diagram definition. Note that the name is the same name referenced in the component.

Line 2 — A catch statement. Rather than coding to the FAILURE event (of KTP 3), the actual exception thrown by an Action can be caught. The catch is similar to a state transition since it defines a result state when the exception is caught.

Lines 3-5 — The component mode "display" is represented in the state diagram. Only one event is supported, the "Go" event, which will cause the transition to the "xSetSymbol" state.

Lines 6-13 — The component mode "xSetSymbol", an action state written in JavaScript that takes the symbol input and places it into the user context.

1.22.4 Device Variations

The stockquote component supports variants in HTML and WML. This is achieved by defining the appropriate variant templates for each of the widgets utilized in rendering the display for the component. The remainder of the component definition is independent of the view, and thus it is independent of the device that it will ultimately be displayed on.

An advantage of using pre-built GUI widgets, such as the Generic Table widget, is that the component you are building will automatically have support for whatever devices and to some extent, styles and locales, that the widget supports, as long as the other widgets and components your component utilizes also offer the same support. If a certain variant is not supported, a new template variant would be required, but the bindings and attributes for the component definition remain unchanged.

The WML variant for the stockquotedisplay widget is listed below.

file: stockquote.kapp

```
<?xml version="1.0"?>
<!DOCTYPE ktp:kapp SYSTEM "http://www.kinzan.net/dtd/kapp.dtd" >

<ktp:kapp name="stockquote" xmlns:ktp="http://www.kinzan.net">

    <!-- domains -->
    <ktp:domainList>
        <ktp:domain name="localhost" documentRoot="stockquote"/>
    </ktp:domainList>

    <ktp:beanManagerList>
        <ktp:beanManager name="kinzan:/beanmanager/kinzan/LocalEJBContainer"/>
        <ktp:beanManager name="kinzan:/beanmanager/kinzan/RemoteEJBContainer"/>
    </ktp:beanManagerList>

    <!-- Includes -->
    <ktp:includeList>
        <ktp:include>simplegui.kapp</ktp:include>
        <ktp:include>genericdatatable.kapp</ktp:include>
    </ktp:includeList>

    <!-- Styles -->
    <ktp:styleList>
        <ktp:style name="stocktable" file="stocktable.style">
            <ktp:description>Styles for the stock table</ktp:description>
        </ktp:style>
    </ktp:styleList>

    <!-- Widgets -->
    <ktp:widgetList>

        <ktp:widget name="StockDisplay">

            <ktp:attributeList>
                 <ktp:local name="prompt" type="java.lang.String"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"
                                locale="false">
                StockDisplayTemplate.kwt
            </ktp:widgetTemplate>

        </ktp:widget>

    </ktp:widgetList>

    <ktp:componentTypeList>

        <!-- The component name must be unique within this site -->
        <ktp:componentType name="StockQuote" defaultMode="display">

            <ktp:description>This is a stock quote component</ktp:description>

            <ktp:attributeList>
                <ktp:local name="title"
                           type="java.lang.String"
                           defaultValue="Stocks"/>
```

```
            <ktp:local name="defaultSymbols"
                 type="java.lang.String"
                 defaultValue="^DJI,^IXIC,^XAX"/>

            <ktp:local name="dataType"
                 type="java.lang.String"
                 defaultValue="symbol,price,change,date"/>

        </ktp:attributeList>

        <ktp:componentList>
            <ktp:component name="symbolInput" type="TextBox"/>
            <ktp:component name="dataDisplay"
                           type="GenericDataTable"
                           style="stocktable"/>
        </ktp:componentList>

        <ktp:modelBeanList>

            <ktp:modelBean name="StockSymbols">
                <ktp:rule name="StockTool" method="getStockSymbols"/>
                <ktp:param name="param1"
                     type="java.lang.String"
                     defaultValue="guest">
                    <ktp:from userContext="accountID"/>
                </ktp:param>

                <ktp:param name="param2" type="java.lang.Vector">
                    <ktp:from attribute="defaultSymbols"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                </ktp:param>

                <ktp:param name="param3" type="java.lang.String" defaultValue="SUNW">
                    <ktp:from userContext="net.kinzan.stockquote.currentSymbol"/>
                </ktp:param>
            </ktp:modelBean>

            <ktp:modelBean name="StockData">
                <ktp:rule name="StockQuote" method="getStockQuotes"/>
                <ktp:param name="param1" type="java.util.Vector">
                    <ktp:from modelBean="StockSymbols"/>
                </ktp:param>
                <ktp:param name="param2" type="java.util.Vector">
                    <ktp:from attribute="dataType"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                </ktp:param>
            </ktp:modelBean>

        </ktp:modelBeanList>

        <ktp:componentModeList>

            <ktp:componentMode name="display" widget="StockDisplay">
                <ktp:zone name="inputArea" component="symbolInput"/>
                <ktp:zone name="displayArea" component="dataDisplay"/>
                <ktp:bindings>

                    <ktp:binding widgetAttribute="prompt">
                        <ktp:from literal="Symbol: "/>
                    </ktp:binding>

                    <ktp:binding component="symbolInput"
                                 attribute="value" defaultValue="">
```

```
                        <ktp:from userContext="net.kinzan.stockquote.currentSymbol"/>
                    </ktp:binding>

                    <ktp:binding component="symbolInput" attribute="size">
                        <ktp:from literal="5"/>
                    </ktp:binding>

                    <ktp:binding component="symbolInput" attribute="name">
                        <ktp:from literal="symbol"/>
                    </ktp:binding>

                    <ktp:binding component="dataDisplay" attribute="content">
                        <ktp:from modelBean="StockData"/>
                    </ktp:binding>
                    <ktp:binding component="dataDisplay" attribute="header">
                        <ktp:from literal="Symbol,Price,Change,Date"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                    </ktp:binding>
                    <ktp:binding component="dataDisplay" attribute="footer">
                        <ktp:from literal="Symbol,Price,Change,Date"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                    </ktp:binding>
                    <ktp:binding component="dataDisplay"
                                 attribute="defaultStyleClass">
                        <ktp:from literal="DefaultTable"/>
                    </ktp:binding>
                    <ktp:binding component="dataDisplay" attribute="rowClasses">
                        <ktp:from literal="GrayTR,WhiteTR,GrayTR,WhiteTR"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                    </ktp:binding>
                    <ktp:binding component="dataDisplay" attribute="headerClasses">
                        <ktp:from literal="HeaderBar"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                    </ktp:binding>
                    <ktp:binding component="dataDisplay" attribute="dataTypes">
                        <ktp:from
literal="symbol,price,price_inc,price_dec,price_unc"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                    </ktp:binding>
                    <ktp:binding component="dataDisplay" attribute="dataTypeClass">
                        <ktp:from
literal="StockSymbol,StockPrice,PriceInc,PriceDec,PriceUnc"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                    </ktp:binding>
                    <ktp:binding component="dataDisplay" attribute="dataTypeReplace">
                        <ktp:from literal=",,,,Unchanged"
transformation="net.kinzan.rendering.runtime.transformation.StringToVector"/>
                    </ktp:binding>

                </ktp:bindings>

            </ktp:componentMode>

        </ktp:componentModeList>

        <ktp:stateDiagram name="Stockquote"/>

    </ktp:componentType>

</ktp:componentTypeList>


<ktp:KPDList>
```

```
        <ktp:KPD name="StockPage" title="Stocks!">
            <ktp:description>A Stock page</ktp:description>
            <ktp:component name="myStockComponent" type="StockQuote"/>
        </ktp:KPD>
    </ktp:KPDList>


</ktp:kapp>
```

end of file: stockquote.kapp

file: stockDisplayTemplate.kwt (html version)

```
<table>
    <tr>
        <td>
            <ktp:zone name="displayArea"/>
        </td>
    </tr>

    <tr>
        <td>
            <ktp:form>
                <ktp:text text="widget.prompt"/>
                <ktp:zone name="inputArea"/>
                <ktp:submit value="Go"/>
            </ktp:form>
        </td>
    </tr>

</table>
```

end of file: stockDisplayTemplate.kwt (html version)

file: stockDisplayTemplate.kwt (wml version)

```
<ktp:zone name="displayArea"/>


<p>
    <ktp:text text="widget.prompt"/>:
    <ktp:zone name="inputArea"/>
</p>
<p align="center">
    <ktp:form>
        <ktp:submit value="Go"/>
    </ktp:form>
</p>
```

end of file: stockDisplayTemplate.kwt (wml version)

file: stockquote.ksd

```
<?xml version="1.0"?>

<!DOCTYPE ktp:stateDiagram SYSTEM "http://www.kinzan.net/dtd/ksd.dtd">

<ktp:stateDiagram name="Stockquote" startState="display"
xmlns:ktp="http://www.kinzan.net">

    <ktp:catch exception="java.lang.Exception" state="display"/>

    <ktp:componentState name="display">
        <ktp:transition event="Go" state="xSetSymbol"/>
    </ktp:componentState>


    <ktp:actionState name="xSetSymbol">
        <ktp:script language="JavaScript">
            symbol = actionEvent.getAttribute( "symbol" );
            userContext.setAttribute( "net.kinzan.stockquote.currentSymbol", symbol );

            actionEvent.setName( "SetSymbol" );
        </ktp:script>
        <ktp:transition event="SetSymbol" state="display"/>
    </ktp:actionState>


</ktp:stateDiagram>
```

end of file: stockquote.ksd

5.Syndicated News Component

1.23 Introduction

1.23.1 Audience

This document is intended for developers who will be working with the Kinzan Technology Platform. It explains some aspects of the KTP 4 component architecture by walking through an example component. This document assumes that the reader has reviewed the weather and stockquote component documents (above) and understands the terms described therein.

1.23.2 Overview

The syndicated news component retrieves news with an EJB session bean. This news is then displayed with a set of widgets. The type and source of news can be selected from a list of news categories and feeds. The new settings for a configured news component are stored in the user's context.

1.24 Syndicated News Component

The news component achieves all its functionality by implementing 3 modes. The first mode, Display, is the default mode. In this mode, the component uses a set of nested components to display an Edit button and a list of news items.

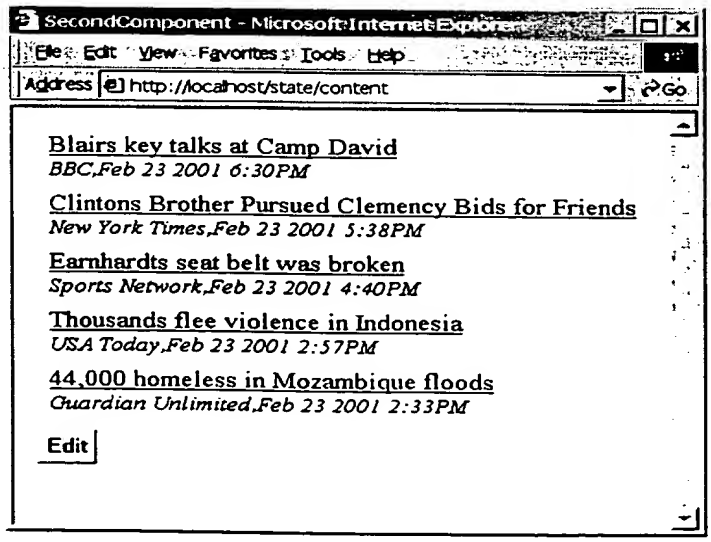


Figure 1

When the Edit button is pressed, the news component transitions to its next mode, Category. In the Category mode, the component presents a drop down lists box where the user can select the type of news category to display.

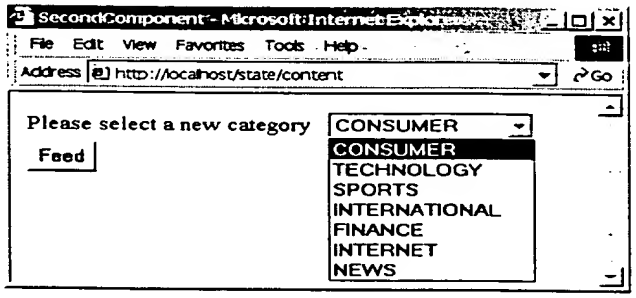


Figure 2

Once a news category is selected, the user can press the Feed button. The Feed button transitions the component to its third and final mode, Feed. In the Feed mode, the user has the opportunity to select a news feed for the previously selected news category.

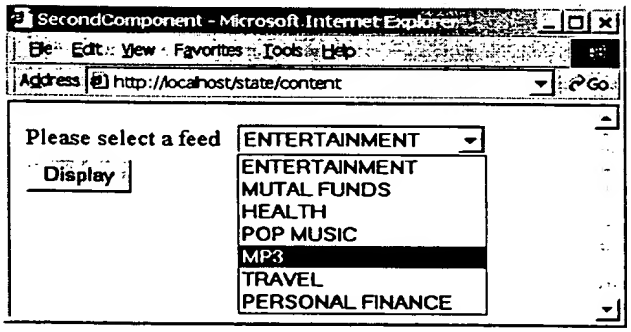


Figure 3

When a Feed is selected, the user then presses the Display button to go back to the Display mode.

The next sections provides a detailed explanation of how the functionality of this component is achieved with the KTP.

1.24.1 Syndicated News Model

The news component uses the ContentFetcher EJB bean as a business rule to get its content. This stateless session bean serves as a wrapper to the Content Service. The Content Service uses CORBA to communicate with a remote

Content Server. Here is the listing for the remote interface of the ContentFetcher bean.

```
public interface ContentFetcher
    extends EJBObject, Remote
{
    public String[] getCategories()
        throws RemoteException;

    public String[] getFeedsFromCategory( String category )
        throws RemoteException;

    public Article[] getArticles( String category, String name)
        throws RemoteException;
}
```

As you can see, the ContentFetcher bean provides three methods: getCategories, getFeedsFromCategories, and getArticles. The getCategories method returns an array of available news categories. The getFeedsFromCategory method retrieves an array of available feeds for a specified category. And, the getArticles method returns the news items for a specified category and feed. These methods will be used in the component Kapp file.

A listing for the implementation of the ContentFetcher bean is provided at the end of this section. This implementation assumes that the ContentFetcher bean is deployed in the local EJB container.

1.24.2 Syndicated News Widget Definitions

The news component employs widgets to create 3 views. Each view maps to a component mode. The first view gets rendered when the component is in Display mode. This view consists of the following 3 widgets: TwoVerticalZones, HyperLinkableParagraph and EmptyForm. These widgets are assembled together to produce the view depicted in Figure 1.

```
1       <ktp:widget name="TwoVerticalZones">
2           <ktp:widgetTemplate style="false" locale="false">
3               twoVerticalZones.kwt
4           </ktp:widgetTemplate>
5       </ktp:widget>

6       <ktp:widget name="HyperLinkableParagraph">
7           <ktp:attributeList>
8               <ktp:local name="linkableParagraphList"
9                          type="com.kinzan.widget.LinkableParagraph[]"/>
10              <ktp:local name="maxParagraphs" type="java.lang.Integer"
11                         defaultValue="10"/>
12              <ktp:local name="bulletGraphics" type="java.lang.String"/>
13          </ktp:attributeList>
```

```
14          <ktp:widgetTemplate style="false" locale="false">
15             hyperLinkableParagraph.kwt
16          </ktp:widgetTemplate>
17      </ktp:widget>

18      <ktp:widget name="EmptyForm">
19          <ktp:attributeList>
20              <ktp:local name="event" type="java.lang.String"/>
21          </ktp:attributeList>

22          <ktp:widgetTemplate style="false" locale="false">
23              emptyForm.kwt
25          </ktp:widgetTemplate>
26      </ktp:widget>
```

Lines 1-5 The TwoVerticalZones widget is defined here. The main function of this widget is to divide the space assigned to it into two vertical zones. The news component uses this widget to place the HyperLinkableParagraph widget in the top zone and the EmptyForm widget in the bottom zone.

Line 6 The HyperLinkableParagraph widget is defined here. This widget displays a list of paragraphs in a table.

Lines 8-9 The HyperLinkableParagraph defines a linkableParagraphList attribute. This attribute is an array of LinkableParagraph objects. A LinkableParagraph object consists of a header, text, and footer. The header, text, and footer can be displayed as a hyperlink if a URL is associated with it.

Lines 10-11 The number of paragraphs displayed can be controlled by setting this attribute.

Line 12 The HyperLinkableParagraph widget can also display an image at the start of each paragraph. This attribute can be used to specify the URL for the image.

Lines 18-26 The EmptyForm widget provides a submit button. The label for this button can be customized by setting the event attribute defined in this widget.

The second and third views (see Figures 2 and 3) are rendered by the DropDownListForm widget. This widget presents a list box and a submit button. In the Category mode, this widget is used to prompt the user for a news category. Moreover, in the Feed mode, the widget is used to display the available feeds for the chosen news category. Here is the definition for the DropDownListForm widget.

```
27      <ktp:widget name="DropDownListForm">

28          <ktp:attributeList>
29              <ktp:local name="prompt" type="java.lang.String"/>
30              <ktp:local name="listPostName" type="java.lang.String"/>
31              <ktp:local name="listSize" type="java.lang.String" defaultValue="1"/>
32              <ktp:local name="options" type="java.lang.String[]"/>
33              <ktp:local name="event" type="java.lang.String"/>
```

```
34          </ktp:attributeList>

35          <ktp:widgetTemplate style="false" locale="false">
36              dropDownListForm.kwt
37          </ktp:widgetTemplate>

38      </ktp:widget>
```

Line 29 The prompt attribute is the string that should be used to prompt the user.

Line 30 The listPostName attribute is the form variable name to use when the list is created.

Line 31 The listSize attribute defines the size of the list box.

Line 32 The options attribute contains an array of strings to display in the list box.

Line 33 The event attribute provides the string that is displayed in the submit button.

1.24.3 Syndicated News Component Definition

As mentioned earlier, the first view consists of 3 widgets. To be able to assemble this view, a component definition is required for each widget that needs to be nested i.e. HyperLinkableParagraph and the EmptyForm widgets. Nesting at the component level results in loosely couple widgets which makes the task of replacing widgets much easier. Here are the component definitions for the EmptyForm and the HyperLinkableParagraph widgets:

```
1       <ktp:componentType name="ParagraphList" defaultMode="display">

2           <ktp:description>
3               A component that contains a list of paragraphs
4           </ktp:description>
5           <ktp:attributeList>
6               <ktp:reference name="linkableParagraphList"
7                              widget="HyperLinkableParagraph"
8                              attribute="linkableParagraphList"/>
9               <ktp:reference name="maxParagraphs"
10                             widget="HyperLinkableParagraph"
11                             attribute="maxParagraphs"/>
12              <ktp:reference name="bulletGraphics"
13                             widget="HyperLinkableParagraph"
14                             attribute="bulletGraphic"/>
15           </ktp:attributeList>

16          <ktp:componentModeList>
17              <ktp:componentMode name="display" widget="HyperLinkableParagraph"/>
18          </ktp:componentModeList>

19      </ktp:componentType>

20      <ktp:componentType name="EmptyForm" defaultMode="display">
21          <ktp:description>An empty form component</ktp:description>
```

```
22          <ktp:attributeList>
23              <ktp:reference name="event" widget="EmptyForm" attribute="event"/>
24          </ktp:attributeList>

25          <ktp:componentModeList>
26              <ktp:componentMode name="display" widget="EmptyForm"/>
27          </ktp:componentModeList>

28      </ktp:componentType>
```

Note that these components are very simple. They expose the widget attributes at the component level. They support only one mode and they do not define a state diagram.

The news component can now be defined:

```
29       <ktp:componentType name="News" defaultMode="display">

30           <ktp:description>This is a very simple news component</ktp:description>

31           <ktp:attributeList>
32               <ktp:local name="title" type="java.lang.String" defaultValue="News"/>
33           </ktp:attributeList>

34           <ktp:componentList>
35               <ktp:component name="list" type="ParagraphList"/>
36               <ktp:component name="form" type="EmptyForm"/>
37           </ktp:componentList>
```

Line 29 The news component is defined here. The default mode is set to Display.

Lines 35-36 Instances of the ParagraphList and EmptyForm components are created and assigned the names "list" and "form"

The news component uses three model beans. Each model bean will be used to provide the data for a different mode. These model beans are generated by the same rule, SyndicatedContent. This rule is mapped to the ContentFetcher bean defined in the previous sections. Here is how all this is done in the content KApp file:

```
38           <ktp:modelBean name="NewsArticles">
39               <ktp:rule name="SyndicatedContent" method="getArticles"/>
40               <ktp:param name="category" type="java.lang.String"
41                          defaultValue="NEWS">
42                  <ktp:from userContext="category"/>
43               </ktp:param>
44               <ktp:param name="feed" type="java.lang.String"
45                          defaultValue="TOP STORIES">
46                  <ktp:from userContext="feed"/>
47               </ktp:param>
```

```
48          </ktp:modelBean>

49          <ktp:modelBean name="Category">
50              <ktp:rule name="SyndicatedContent" method="getCategories"/>
51          </ktp:modelBean>

52          <ktp:modelBean name="Feed">
53              <ktp:rule name="SyndicatedContent" method="getFeedsFromCategory"/>
54              <ktp:param name="category" type="java.lang.String"
55                         defaultValue="TECHNOLOGY">
56                 <ktp:from userContext="category" />
57              </ktp:param>
58          </ktp:modelBean>
```

Lines 38-48 The NewsArticles model bean is obtained by calling the getArticles method on the ContentFetcher bean i.e. SyndicatedContent rule. Note that the category and feed parameters are taken from the user's context.

Lines 49-51 The Category model bean is also obtained by using the SyndicatedContent rule. This time, we call the getCategories method.

Lines 52-58 The Feed model contains the feeds for the provided category. The category is obtained from the user's context.

The Display mode for the news component gets its contents from the NewsArticles model bean and displays its contents using the TwoVerticalZones widget and nested components.

```
59          <ktp:componentMode name="display" widget="TwoVerticalZones">

60              <ktp:zone name="top" component="list"/>
61              <ktp:zone name="bottom" component="form"/>

62              <ktp:bindings>

63                  <ktp:binding component="list" attribute="linkableParagraphList" >
64                     <ktp:from modelBean="NewsArticles"
transformation="com.kinzan.example.content.transformation.ArticlesToLinkableParagraphs"/>

65                  </ktp:binding>

66                  <ktp:binding component="list" attribute="maxParagraphs"
67                               defaultValue="5">
68                      <ktp:from userContext="net.kinzan.component.maxparagraphs"/>
69                  </ktp:binding>

70                  <ktp:binding component="form" attribute="event">
71                      <ktp:from literal="Edit"/>
72                  </ktp:binding>

73                  <ktp:binding attribute="title">
74                      <ktp:from literal="News"/>
75                  </ktp:binding>

76              </ktp:bindings>
```

```
77          </ktp:componentMode>
```

Lines 59-61 The Display mode uses the TwoVerticalZones widget as its main view. The "list" and "form" components are mapped to the top and bottom zones.

Lines 63-65 The linkableParagraphList attribute in the "list" component is bound to the NewsArticles model bean. Since the NewsArticles model bean is obtained from invoking the getArticles method on the ContentFetcher bean, the model bean type is an array of Article objects. We need to transform this array of Articles to an array of LinkableParagraph objects. This is done by specifying the com.kinzan.example.content.transformation.ArticlesToLinkableParagraphs transformation.

Lines 66-69 By default, the maximum number of paragraphs to display is set to 5 unless this value is specified in the user's context.

Lines 70-72 The event attribute of the "form" component is set to "Edit". This is the event that will be passed to the State Manager when the submit button is pressed.

The Category mode uses the DropDownListForm widget for its view. The data is obtained from the Category model bean. Unlike the previous mode, this mode does not contain nested components.

```
78          <ktp:componentMode name="category" widget="DropDownListForm">
79              <ktp:bindings>

80                  <ktp:binding widgetAttribute="prompt" >
81                      <ktp:from literal="Please select a new category "/>
82                  </ktp:binding>

83                  <ktp:binding widgetAttribute="listPostName">
84                      <ktp:from literal="category"/>
85                  </ktp:binding>

86                  <ktp:binding widgetAttribute="listSize">
87                      <ktp:from literal="1"/>
88                  </ktp:binding>

89                  <ktp:binding widgetAttribute="event">
90                      <ktp:from literal="Feed"/>
91                  </ktp:binding>

92                  <ktp:binding widgetAttribute="options">
93                      <ktp:from modelBean="Category"/>
94                  </ktp:binding>

95                  <ktp:binding attribute="title">
96                      <ktp:from literal="Select Category"/>
97                  </ktp:binding>
98              </ktp:bindings>
99          </ktp:componentMode>
```

Line 78 The Category mode is defined and associated with the DropDownListForm.

Lines 83-85 The selected the list box item is assigned the form name "category". This name will be used later on in the state diagram.

Lines 89-91 The next event is set to "Feed".

Lines 92-94 The list box is populated with the "Category" model bean.

The definition of the Feed mode is similar to the Category mode. The data comes from the Feed model bean and the next event is set to "Display". Please see the content.KApp file for further information.

To be able to transition from the different component modes and process the requests, the news component needs a state diagram. This is how it is defined for the news component.

```
<ktp:stateDiagram name="Content"/>
```

1.24.4 News Component State Diagram

The news component defines three component modes. Each component mode is associated with an action state. The action state executes the appropriate edits and transitions the component to its next mode. Since the nested components do not define the corresponding states, all events are handled by the news component state diagram. In this example, all actions are implemented in JavaScript.

```
1   <ktp:stateDiagram name="Content" startState="display"
2                     xmlns:ktp="http://www.kinzan.net">

3   <ktp:catch exception="java.lang.Exception" state="display"/>

4   <ktp:componentState name="display">
5       <ktp:transition event="Edit" state="category"/>
6   </ktp:componentState>

7   <ktp:componentState name="category">
8       <ktp:transition event="Feed" state="xSetCategory"/>
9   </ktp:componentState>

10  <ktp:componentState name="feed">
11      <ktp:transition event="Display" state="xSetDisplay"/>
12  </ktp:componentState>


13  <ktp:actionState name="xSetCategory">
14      <ktp:script language="JavaScript">
```

```
15                  userContext.setAttribute( "category",
16                                            actionEvent.getAttribute( "category" ) );
17                  actionEvent.setName( "setCategory" );
17              </ktp:script>
18              <ktp:transition event="setCategory" state="feed"/>

19          </ktp:actionState>

20          <ktp:actionState name="xSetDisplay">
21              <ktp:script language="JavaScript">
22                  userContext.setAttribute( "feed",
23                                            actionEvent.getAttribute( "feed" ) );
34                  actionEvent.setName( "setFeed" );
35              </ktp:script>
36              <ktp:transition event="setFeed" state="display"/>
37          </ktp:actionState>
38      </ktp:stateDiagram>
```

Lines 13-18 The xSetCategory action state is executed when the news component is in the Category component mode and the "Feed" button is pressed. The action takes the form variable posted with the name "category" and sets its value into the user context. It then generates a "setCategory" event which transitions the news component to the Feed component mode.

Lines 20-37 The xSetDisplay action state is executed when the news component is in the Feed component mode and the "Display" button is pressed. This time, the form variable "feed" is set into the user context and the component is transitioned to the Display mode.

```
<?xml version="1.0"?>
<!DOCTYPE ktp:kapp SYSTEM "http://www.kinzan.net/dtd/kapp.dtd" >

<ktp:kapp name="content" xmlns:ktp="http://www.kinzan.net">

    <!-- Define domains -->
    <ktp:domainList>
        <ktp:domain name="@content.domain@" documentRoot="content"/>
    </ktp:domainList>

    <ktp:beanManagerList>
        <ktp:beanManager name="kinzan:/beanmanager/kinzan/LocalEJBContainer"/>
    </ktp:beanManagerList>

    <!-- Includes -->
    <ktp:includeList>
        <ktp:include>simplegui.kapp</ktp:include>
        <ktp:include>stylelib.kapp</ktp:include>
    </ktp:includeList>


    <!-- Widgets -->
    <ktp:widgetList>

        <!--A generic list of paragraph, each paragraph may consists of a header,
            header link, text, text link, footer, and footer link.  All links are
            optional.  If a link is supplied the associated element is made a
            hyperlink.  The maxParagraphs attribute limits the number of paragraph
            displayed by this list.  The bulletGraphic attribute can be set
            to a name of a gif (Note: It should be an asset) -->
        <ktp:widget name="HyperLinkableParagraph">
            <ktp:attributeList>
                <ktp:local name="linkableParagraphList"
                           type="com.kinzan.widget.LinkableParagraph[]"/>
                <ktp:local name="maxParagraphs" type="java.lang.Integer"
                           defaultValue="5"/>
                <ktp:local name="bulletGraphics" type="java.lang.String"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false"locale="false">
                hyperLinkableParagraph.kwt
            </ktp:widgetTemplate>
        </ktp:widget>

        <!-- Dropdown list form -->
        <ktp:widget name="DropDownListForm">

            <ktp:attributeList>
                <ktp:local name="prompt" type="java.lang.String"/>
                <ktp:local name="listPostName" type="java.lang.String"/>
                <ktp:local name="listSize" type="java.lang.String" defaultValue="1"/>
                <ktp:local name="options" type="java.lang.String[]"/>
                <ktp:local name="event" type="java.lang.String"/>
            </ktp:attributeList>

            <ktp:widgetTemplate style="false" locale="false">
                dropDownListForm.kwt
            </ktp:widgetTemplate>

        </ktp:widget>
```

```
    <!-- An empty form widget -->
    <ktp:widget name="EmptyForm">
        <ktp:attributeList>
            <ktp:local name="event" type="java.lang.String"/>
        </ktp:attributeList>

        <ktp:widgetTemplate style="false" locale="false">
            emptyForm.kwt
        </ktp:widgetTemplate>
    </ktp:widget>

    <!-- Two zone widget -->
    <ktp:widget name="TwoVerticalZones">
        <ktp:widgetTemplate style="false" locale="false">
            twoVerticalZones.kwt
        </ktp:widgetTemplate>
    </ktp:widget>


</ktp:widgetList>

<ktp:componentTypeList>

    <!-- A paragraph list component -->
    <ktp:componentType name="ParagraphList" defaultMode="display">

        <ktp:description>
            A component that contains a list of paragraphs
        </ktp:description>

        <ktp:attributeList>
            <ktp:reference name="linkableParagraphList"
                           widget="HyperLinkableParagraph"
                           attribute="linkableParagraphList"/>
            <ktp:reference name="maxParagraphs" widget="HyperLinkableParagraph"
                           attribute="maxParagraphs"/>
            <ktp:reference name="bulletGraphics" widget="HyperLinkableParagraph"
                           attribute="bulletGraphic"/>
        </ktp:attributeList>

        <ktp:componentModeList>
            <ktp:componentMode name="display" widget="HyperLinkableParagraph"/>
        </ktp:componentModeList>

    </ktp:componentType>

    <!-- An empty form component -->
    <ktp:componentType name="EmptyForm" defaultMode="display">
        <ktp:description>An empty form component</ktp:description>

        <ktp:attributeList>
            <ktp:reference name="event" widget="EmptyForm" attribute="event"/>
        </ktp:attributeList>

        <ktp:componentModeList>
            <ktp:componentMode name="display" widget="EmptyForm"/>
        </ktp:componentModeList>

    </ktp:componentType>


    <!-- Syndicated News component -->
    <ktp:componentType name="News" defaultMode="display">
```

```
<ktp:description>This is a very simple news component</ktp:description>

<ktp:attributeList>
    <ktp:local name="title" type="java.lang.String" defaultValue="News"/>
</ktp:attributeList>

<!-- Nested components -->
<ktp:componentList>
    <ktp:component name="list" type="ParagraphList"/>
    <ktp:component name="form" type="EmptyForm"/>
</ktp:componentList>

 <!-- Model bean list -->
<ktp:modelBeanList>

     <!-- Call syndicated content session bean to get News items -->
     <ktp:modelBean name="NewsArticles">
         <ktp:rule name="SyndicatedContent" method="getArticles"/>

         <ktp:param name="category" type="java.lang.String"
                    defaultValue="NEWS">
            <ktp:from userContext="category"/>
         </ktp:param>

         <ktp:param name="feed" type="java.lang.String"
                    defaultValue="TOP STORIES">
            <ktp:from userContext="feed"/>
         </ktp:param>

     </ktp:modelBean>

     <!-- Call the syndicated content session bean to get a list
          of categories -->
     <ktp:modelBean name="Category">
         <ktp:rule name="SyndicatedContent" method="getCategories"/>
     </ktp:modelBean>

     <ktp:modelBean name="Feed">
         <ktp:rule name="SyndicatedContent" method="getFeedsFromCategory"/>
         <ktp:param name="category" type="java.lang.String"
                    defaultValue="TECHNOLOGY">
            <ktp:from userContext="category" />
         </ktp:param>
     </ktp:modelBean>

</ktp:modelBeanList>


<!-- Modes -->
<ktp:componentModeList>

    <!-- Display -->
    <ktp:componentMode name="display" widget="TwoVerticalZones">

        <!-- Map the paragraph list and the empty from into the
             zones for this widget -->
        <ktp:zone name="top" component="list"/>
        <ktp:zone name="bottom" component="form"/>

        <!-- bindings for these two components -->
        <ktp:bindings>
```

```
                        <ktp:binding component="list" attribute="linkableParagraphList" >
                            <ktp:from modelBean="NewsArticles"
transformation="com.kinzan.example.content.transformation.ArticlesToLinkableParagraphs"/>
                        </ktp:binding>

                        <ktp:binding component="list" attribute="maxParagraphs"
                                     defaultValue="5">
                            <ktp:from userContext="net.kinzan.component.maxparagraphs"/>
                        </ktp:binding>

                        <ktp:binding component="form" attribute="event">
                            <ktp:from literal="Edit"/>
                        </ktp:binding>

                        <ktp:binding attribute="title">
                            <ktp:from literal="News"/>
                        </ktp:binding>

                    </ktp:bindings>

                </ktp:componentMode>

                <!-- Select news category mode -->
                <ktp:componentMode name="category" widget="DropDownListForm">
                    <ktp:bindings>

                        <ktp:binding widgetAttribute="prompt" >
                            <ktp:from literal="Please select a new category "/>
                        </ktp:binding>

                        <ktp:binding widgetAttribute="listPostName">
                            <ktp:from literal="category"/>
                        </ktp:binding>

                        <ktp:binding widgetAttribute="listSize">
                            <ktp:from literal="1"/>
                        </ktp:binding>

                        <ktp:binding widgetAttribute="event">
                            <ktp:from literal="Feed"/>
                        </ktp:binding>

                        <ktp:binding widgetAttribute="options">
                            <ktp:from modelBean="Category"/>
                        </ktp:binding>

                        <ktp:binding attribute="title">
                            <ktp:from literal="Select Category"/>
                        </ktp:binding>

                    </ktp:bindings>
                </ktp:componentMode>

                <!-- Select news feed mode -->
                <ktp:componentMode name="feed" widget="DropDownListForm">
                    <ktp:bindings>

                        <ktp:binding widgetAttribute="prompt" >
                            <ktp:from literal="Please select a feed "/>
                        </ktp:binding>

                        <ktp:binding widgetAttribute="listPostName">
                            <ktp:from literal="feed"/>
```

```
                    </ktp:binding>

                    <ktp:binding widgetAttribute="listSize">
                        <ktp:from literal="1"/>
                    </ktp:binding>

                    <ktp:binding widgetAttribute="event">
                        <ktp:from literal="Display"/>
                    </ktp:binding>

                    <ktp:binding widgetAttribute="options">
                        <ktp:from modelBean="Feed"/>
                    </ktp:binding>

                    <ktp:binding attribute="title">
                        <ktp:from literal="Select Feed"/>
                    </ktp:binding>

                </ktp:bindings>
            </ktp:componentMode>


        </ktp:componentModeList>

        <!-- Define the component state diagram -->
        <ktp:stateDiagram name="Content"/>

    </ktp:componentType>

</ktp:componentTypeList>

<ktp:KPDList>
    <ktp:KPD name="secondComponent" title="SecondComponent">
        <ktp:description>My Second component</ktp:description>
        <ktp:component name="contentComponent" type="News"/>
    </ktp:KPD>
</ktp:KPDList>


</ktp:kapp>

end of file: content.kapp
```

file: ContentFetcherBean.java

```
public class ContentFetcherBean implements SessionBean
{

    /**
     * The EJB session context.
     */
    private SessionContext iContext;

    /**
     * Gets the news categories that are available from the syndicated news
     * server.
     *
     * @return an array of strings containing the news categories that are
     * avialable.
     * @exception ServiceException if the services encounters a problem in
     * performing the request.
     * @exception LookupException if the service manager can't find the service.
     */
    public String[] getCategories()
        throws ServiceException, LookupException
    {
        AppContext appContext = KinzanApp.GetAppContext();
        RemoteContentService service =
            (RemoteContentService) appContext.getService( RemoteContentService.class );

        return service.getCategories();
    }

    /**
     * Gets the feeds that are available for a given news category.
     *
     * @return an array of strings containing the name of the feeds.
     * @exception ServiceException if the services encounters a problem in
     * performing the request.
     * @exception LookupException if the service manager can't find the service.
     */
    public String[] getFeedsFromCategory( String category )
        throws ServiceException, LookupException
    {
        String[] feedsName = null;

        AppContext appContext = KinzanApp.GetAppContext();
        RemoteContentService service =
            (RemoteContentService) appContext.getService( RemoteContentService.class );

        Feed[] feeds = service.getFeedsFromCategory( category );
        if ( feeds != null )
        {
            feedsName = new String[ feeds.length ];
            for ( int i = 0; i < feeds.length; i++ )
            {
                feedsName[ i ] = feeds[ i ].iName;
            }
        }

        return feedsName;
    }
```

```
    /**
     * Gets the news articles given a category and its associated feed.
     *
     * @return an array of Article objects.
     * @exception ServiceException if the services encounters a problem in
     * performing the request.
     * @exception LookupException if the service manager can't find the service.
     */
    public Article[] getArticles( String category, String name)
        throws ServiceException, LookupException
    {

        AppContext appContext = KinzanApp.GetAppContext();
        RemoteContentService service =
            (RemoteContentService) appContext.getService( RemoteContentService.class );

        return service.getArticles( category, name );
    }

    /**
     * Callback for the ejb container to notify that the bean is being activate.
     */
    public void ejbActivate()
    {
    }

    /**
     * Callback for the ejb container to notify the bean that is being removed.
     */
    public void ejbRemove()
    {
    }

    /**
     * Callback for the ejb container to notify the bean that is being passivated.
     */
    public void ejbPassivate()
    {
    }

    /**
     * The EJB containt calls this method to set the ejb session.
     */
    public void setSessionContext( SessionContext ctx )
    {
        iContext = ctx;
    }

    /**
     * The EJB container calls this method to notify the bean that it's being
     * created.
     */
    public void ejbCreate()
    {
    }

}
```

end of file: ContentFetcherBean.java

6.Kinzan Glossary

Term	Meaning
Action	Performs the appropriate e-business rule on behalf of the user.
Adaptive Web Service	An autonomous business object that logically couples presentation, code, and data and thus encapsulates a discrete functionality that can be dynamically loaded at run time to add that functionality to an Internet or web based application.
Assets	Leaf objects that are included on a page via widgets. They include but are not limited to JSP, TXT, HTML, and image files.
Branded Assets	Assets with variants that can be resolved dynamically based on the current style, locale, and/or target device.
Components	A logical object or Mini-Application, coupling the model, view, and controller for the user.
Containers	Collections of content and presentation, typically deployed within a page layout. Containers may contain components (a.k.a. "Adaptive web services") that contain widgets, JSP fragments, other containers, links to external resources, or any of these in combination.
Controller Layer	In MVC architecture, the means by which the user provides input or changes the state of the model. In KTP, the controller layer comprises the state manager, state diagrams, and actions.
KApp File (Kinzan Application File)	An XML file that describes the site and the assets that belong to it and is the equivalent of a Makefile.
KTP	Kinzan Technology Platform is a framework that easily allows for modular construction of Internet Applications.
KPD (Kinzan Page Descriptor)	A collection of references to modules that represent style, structure, presentation, content, and branded assets. Page configuration and assembly instructions are captured as XML in the KPD file and used to intelligently drive the rendering process.
KSD	An XML file that describes the state flow for the

Term	Meaning
(Kinzan State Diagram)	component.
KWT (Kinzan Widget Template)	A definition structure for the widgets, including zones (placeholders) for content within that structure.
Localization	The elements that make up a KPD collection may need the capability to support final JSP's for multiple languages. Localization is the method of making your application support various countries, languages, and dialects at the logic level so that it handles single byte, double byte, currency, etc.
Presentation	Within each zone of a page layout, various display elements are collected into a presentation for that zone. Presentation is modeled separately because similar content may be presented in different ways in different contexts. For example, sales data may be presented in graphical form or in a spreadsheet form without changing the content.
Rendering Service	Creates the response or resulting html that the user consumes by digesting the style, Kinzan Page Descriptor, widget and bound data.
Service	Is a representation of different ways to access information and to store information. These include databases, messaging services, higher-level application components like EJB's, etc. Typically, the main challenges in working with services are finding the service that provides the information you need, then interfacing with that service to retrieve the information.
Services Manager	Provides an abstraction layer for many kinds of back end services and includes the ability to dynamically look up and bind to services via name lookup servers.
State Diagram	Represents discrete steps in an application flow, with decision points that trigger transitions to different sections of the application.
State Manager	Provides basic transaction management across action components, and facilitates links to model beans, business rules and the Services Manager.
Style	An XML file that defines style elements to apply to pages that reference the style. The format of the style file closely follows that used in Cascading Style Sheets (CSS), specifying a list of tags and the style attributes for each tag. Multiple classes may be defined in a style.

Term	Meaning
	These classes allow you to define variations of style attributes for given tags.
Template	Collections of common resources and content that apply on a site-wide basis. Templates are usually used to contain all "brand able" aspects of an offering, so that an application may be rapidly private-labeled for different partners, and customers may select from multiple branding motifs for their site.
User Context	An object of class userContext that tracks the user's progress through a site. Information that the userContext object contains includes the context ID, the user's current state, and the return stack (the user's movement between actions).
Widget	A generic or custom control that is purely visual.

7.DTD Files

Following are listings for the DTD files for KApp, KSD, KSec, actionValidator, and openejb_conf files.

File: kapp.dtd

```
<!ELEMENT ktp:description (#PCDATA)>
<!ELEMENT ktp:KPD (ktp:description,ktp:component)>
<!ATTLIST ktp:KPD
 name CDATA #REQUIRED
 title CDATA #REQUIRED
 visibility CDATA #IMPLIED
 overwriteable CDATA #IMPLIED
>
<!ELEMENT ktp:KPDList (ktp:KPD+)>
<!ELEMENT ktp:asset (ktp:description?,ktp:variant+)>
<!ATTLIST ktp:asset
 name CDATA #REQUIRED
 overwriteable CDATA #IMPLIED
 visibility CDATA #IMPLIED
>
<!ELEMENT ktp:attributeList ((ktp:local+|ktp:reference+)+)>
<!ELEMENT ktp:beanManager EMPTY>
<!ATTLIST ktp:beanManager
 name CDATA #REQUIRED
>
<!ELEMENT ktp:beanManagerList (ktp:beanManager+)>
<!ELEMENT ktp:binding (ktp:from)>
<!ATTLIST ktp:binding
 attribute CDATA #IMPLIED
 component CDATA #IMPLIED
 defaultValue CDATA #IMPLIED
 widgetAttribute CDATA #IMPLIED
 overwriteable CDATA #IMPLIED
>
<!ELEMENT ktp:bindings (ktp:binding+)>
<!ELEMENT ktp:component (ktp:componentList?,ktp:componentModeList?)?>
<!ATTLIST ktp:component
 reference CDATA #IMPLIED
 name CDATA #IMPLIED
 type CDATA #IMPLIED
 style CDATA #IMPLIED
 uniqueName CDATA #IMPLIED
 overwriteable CDATA #IMPLIED
 visibility CDATA #IMPLIED
>
<!ELEMENT ktp:componentList (ktp:component+)>
<!ELEMENT ktp:componentMode (ktp:zone*,ktp:bindings?)?>
<!ATTLIST ktp:componentMode
 name CDATA #REQUIRED
 widget CDATA #IMPLIED
>
<!ELEMENT ktp:componentModeList (ktp:componentMode+)>
<!ELEMENT ktp:componentType
(ktp:description?,(ktp:attributeList?|ktp:modelBeanList?|ktp:componentList?)*,ktp:compone
ntModeList,ktp:stateDiagram?)>
```

```
<!ATTLIST ktp:componentType
 defaultMode CDATA #REQUIRED
 name CDATA #REQUIRED
 style CDATA #IMPLIED
 visibility CDATA #IMPLIED
 overwriteable CDATA #IMPLIED
 extends CDATA #IMPLIED
>
<!ELEMENT ktp:componentTypeList (ktp:componentType+)>
<!ELEMENT ktp:domain EMPTY>
<!ATTLIST ktp:domain
 documentRoot CDATA #REQUIRED
 name CDATA #REQUIRED
>
<!ELEMENT ktp:domainList (ktp:domain+)>
<!ELEMENT ktp:from EMPTY>
<!ATTLIST ktp:from
 component CDATA #IMPLIED
 attribute CDATA #IMPLIED
 bundle CDATA #IMPLIED
 literal CDATA #IMPLIED
 userContext CDATA #IMPLIED
 modelBean CDATA #IMPLIED
 transformation CDATA #IMPLIED
>
<!ELEMENT ktp:include (#PCDATA)>
<!ELEMENT ktp:includeList (ktp:include+)>
<!ELEMENT ktp:kapp
(ktp:domainList,ktp:beanManagerList?,ktp:includeList?,ktp:styleList?,(ktp:assetList?|ktp:
widgetList?|ktp:componentTypeList?|ktp:componentList?|ktp:KPDList?)*)>
<!ATTLIST ktp:kapp
 name CDATA #REQUIRED
 visibility CDATA #IMPLIED
 overwriteable CDATA #IMPLIED
>
<!ELEMENT ktp:local EMPTY>
<!ATTLIST ktp:local
 defaultValue CDATA #IMPLIED
 name CDATA #REQUIRED
 type CDATA #REQUIRED
 required CDATA #IMPLIED
>
<!ELEMENT ktp:modelBean (ktp:rule,ktp:param*)>
<!ATTLIST ktp:modelBean
 name CDATA #REQUIRED
 scope CDATA #IMPLIED
>
<!ELEMENT ktp:modelBeanList (ktp:modelBean+)>
<!ELEMENT ktp:param (ktp:from)>
<!ATTLIST ktp:param
 defaultValue CDATA #IMPLIED
 name CDATA #REQUIRED
 type CDATA #REQUIRED
>
<!ELEMENT ktp:reference EMPTY>
<!ATTLIST ktp:reference
 attribute CDATA #IMPLIED
 name CDATA #IMPLIED
 widget CDATA #IMPLIED
 component CDATA #IMPLIED
 defaultValue CDATA #IMPLIED
 required CDATA #IMPLIED
>
```

```
<!ELEMENT ktp:rule EMPTY>
<!ATTLIST ktp:rule
 method CDATA #REQUIRED
 name CDATA #REQUIRED
>
<!ELEMENT ktp:stateDiagram EMPTY>
<!ATTLIST ktp:stateDiagram
 name CDATA #REQUIRED
>
<!ELEMENT ktp:style (ktp:description?)>
<!ATTLIST ktp:style
 file CDATA #REQUIRED
 name CDATA #REQUIRED
 visibility CDATA #IMPLIED
 overwriteable CDATA #IMPLIED
>
<!ELEMENT ktp:styleList (ktp:style+)>
<!ELEMENT ktp:widget
(ktp:description?,ktp:attributeList?,ktp:widgetAssetList?,ktp:widgetTemplate)>
<!ATTLIST ktp:widget
 name CDATA #IMPLIED
 visibility CDATA #IMPLIED
>
<!ELEMENT ktp:widgetAssetList (ktp:widgetAsset+)>
<!ELEMENT ktp:widgetList (ktp:widget+)>
<!ELEMENT ktp:widgetTemplate (#PCDATA)>
<!ATTLIST ktp:widgetTemplate
 locale CDATA "true"
 style CDATA "true"
 target CDATA "true"
>
<!ELEMENT ktp:zone EMPTY>
<!ATTLIST ktp:zone
 component CDATA #REQUIRED
 name CDATA #REQUIRED
>
<!ELEMENT ktp:variant (#PCDATA)>
<!ATTLIST ktp:variant
 locale CDATA "true"
 style CDATA "true"
 target CDATA "true"
>
<!ELEMENT ktp:assetList (ktp:asset+)>
<!ELEMENT ktp:widgetAsset EMPTY>
<!ATTLIST ktp:widgetAsset
 asset CDATA #REQUIRED
 name CDATA #REQUIRED
 overwriteable CDATA #IMPLIED
>

end of file: kapp.dtd
```

file: ksd.dtd

```
<!ELEMENT ktp:stateDiagram
(ktp:actionState*,ktp:displayState*,ktp:componentState*,ktp:transition*,ktp:catch*)*>
<!ATTLIST ktp:stateDiagram
          name CDATA #REQUIRED
          startState CDATA #IMPLIED
          directAccess  (true|false) 'true'
          visibility (public|private|final) 'public'>

<!ELEMENT ktp:displayState (ktp:transition*,ktp:property*)*>
<!ATTLIST ktp:displayState
          name CDATA #REQUIRED
          forward CDATA #IMPLIED
          redirect CDATA #IMPLIED
          kpd CDATA #IMPLIED
          permission CDATA #IMPLIED
          directAccess  (true|false) 'true'
          visibility (public|private|final) 'public'>

<!ELEMENT ktp:componentState (ktp:transition*,ktp:property*)*>
<!ATTLIST ktp:componentState
          name CDATA #REQUIRED
          permission CDATA #IMPLIED
          directAccess  (true|false) 'true'
          rethrowEvent (true|false) 'false'
          visibility (public|private|final) 'public'>

<!ELEMENT ktp:actionState (ktp:transition*,ktp:catch*,ktp:property*,ktp:script?)*>
<!ATTLIST ktp:actionState
          name CDATA #REQUIRED
          action CDATA #IMPLIED
          script CDATA #IMPLIED
          permission CDATA #IMPLIED
          directAccess  (true|false) 'true'
          visibility (public|private|final) 'public'>

<!ELEMENT ktp:script (#PCDATA)>
<!ATTLIST ktp:script
          language CDATA #REQUIRED>

<!ELEMENT ktp:property EMPTY>
<!ATTLIST ktp:property
          name CDATA #REQUIRED
          value CDATA #REQUIRED>

<!ELEMENT ktp:transition (ktp:property*)>
<!ATTLIST ktp:transition
          event CDATA #REQUIRED
          state CDATA #REQUIRED
          wizard CDATA #IMPLIED
          clearContext (true|false) 'false'>

<!ELEMENT ktp:catch EMPTY>
<!ATTLIST ktp:catch
          exception CDATA #REQUIRED
          state CDATA #REQUIRED
          wizard CDATA #IMPLIED>
```

end of file: ksd.dtd

file: ksec.dtd

```
<!ELEMENT ktp:ACL
   EMPTY>

<!ATTLIST ktp:ACL
componentUniqueName CDATA #IMPLIED>

<!ELEMENT ktp:appRole
   (ktp:role+)>
<!ELEMENT ktp:componentACL
   (ktp:ACL+)>
<!ELEMENT ktp:componentAccess
   EMPTY>


<!ATTLIST ktp:componentAccess
allaccess CDATA #IMPLIED>

<!ATTLIST ktp:componentAccess
componentUniqueName CDATA #IMPLIED>
<!ELEMENT ktp:componentPermission
   (ktp:permission?)>

<!ATTLIST ktp:componentPermission
allpermission CDATA #IMPLIED>

<!ATTLIST ktp:componentPermission
componentUniqueName CDATA #IMPLIED>
<!ELEMENT ktp:ksec
   (ktp:componentACL?,ktp:appRole)>

<!ATTLIST ktp:ksec
          xmlns:ktp CDATA #IMPLIED
          name CDATA #REQUIRED>
<!ELEMENT ktp:permission
   EMPTY>

<!ATTLIST ktp:permission
name CDATA #IMPLIED>
<!ELEMENT ktp:role
   (ktp:componentAccess*,ktp:componentPermission*)>

<!ATTLIST ktp:role
name CDATA #IMPLIED>
```

end of file: ksec.dtd

file: actionValidator.dtd

```
<?xml encoding="US-ASCII"?>

<!ELEMENT ktp:actionValidator (ktp:parameterSet*,ktp:parameter*)*>
<!ATTLIST ktp:actionValidator
          class CDATA #IMPLIED>

<!ELEMENT ktp:parameterSet (ktp:parameter*)>
<!ATTLIST ktp:parameterSet
          name CDATA #REQUIRED>

<!ELEMENT ktp:parameter (ktp:validator*)>
<!ATTLIST ktp:parameter
          name CDATA #REQUIRED
          nullOkay (true|false) 'false'>

<!ELEMENT ktp:validator (ktp:property*)>
<!ATTLIST ktp:validator
          type CDATA #REQUIRED>

<!ELEMENT ktp:property EMPTY>
<!ATTLIST ktp:property
          name CDATA #REQUIRED
          value CDATA #REQUIRED>
```

end of file: actionValidator.dtd

file: openejb_config.dtd

```
<?xml encoding="US-ASCII"?>

<!-- $Id: openejb_config.dtd,v 1.20 2000/12/16 14:55:29 rmonson Exp $ -->


<!ELEMENT entity-bean (description?, display-name?, small-icon?,large-icon?, ejb-
deployment-id, home, remote, ejb-class, primary-key, jndi-enc?, security-role-ref*)>
<!ELEMENT entity-container (description?, display-name?, container-name, properties?,
entity-bean+)>
<!ELEMENT codebase (#PCDATA)>
<!ELEMENT class-name (#PCDATA)>
<!ELEMENT connection (properties)>

<!ELEMENT connectors (connector+, connection-manager+)>
<!ELEMENT connector (connector-id, connection-manager-id, managed-connection-factory)>
<!ELEMENT connector-id (#PCDATA)>
<!ELEMENT connection-manager-id (#PCDATA)>
<!ELEMENT connection-manager (connection-manager-id, class-name, properties?)>
<!ELEMENT managed-connection-factory (class-name, properties?)>

<!ELEMENT containers    (stateful-session-container|stateless-session-container|entity-
container)+>
<!ELEMENT container-name (#PCDATA)>
<!ELEMENT container-system (containers, security-role+, method-permission+, method-
transaction+)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT display-name (#PCDATA)>
<!ELEMENT ejb-class (#PCDATA)>
<!ELEMENT ejb-deployment-id (#PCDATA)>
<!ELEMENT ejb-ref (ejb-ref-name, home, ejb-ref-location)>
<!ELEMENT ejb-ref-location (ejb-deployment-id | connection)>
<!ELEMENT ejb-ref-name (#PCDATA)>
<!ELEMENT home (#PCDATA)>
<!ELEMENT env-entry (env-entry-name, env-entry-type, env-entry-value)>
<!ELEMENT env-entry-name (#PCDATA)>
<!ELEMENT env-entry-type (#PCDATA)>
<!ELEMENT env-entry-value (#PCDATA)>
<!ELEMENT facilities (intra-vm-server, connectors?, services)>
<!ELEMENT factory-class (#PCDATA)>
<!ELEMENT intra-vm-server (proxy-factory, codebase, properties?)>
<!ELEMENT jndi-enc (env-entry*, ejb-ref*, resource-ref*)>
<!ELEMENT large-icon (#PCDATA)>
<!ELEMENT logical-role-name (#PCDATA)>

<!--
The method element is used to denote a method of an enterprise bean's
home or remote interface, or a set of methods. The ejb-deployment-id element
must be the id of one of the enterprise beans declared in the
container-system; the optional method-intf element allows to
distinguish between a method with the same signature that is defined in
both the home and remote interface; the method-name element specifies
the method name; and the optional method-params elements identify a
single method among multiple methods with an overloaded method name.

There are three possible styles of the method element syntax:

1.    <method>
        <method-name>*</method-name>
      </method>
```

```
   This style is used to refer to all the methods of all
   deployments (home and remote interfaces) in a container-system.

2.    <method>
        <method-name>METHOD</method-name>
      </method>
   This style is used to refer to the specified method in all
   deployments in the container-system. If there are multiple methods with
   the same overloaded name (home and remote interfaces), the element of
   this style refers to all the methods with the overloaded name.

3.    <method>
        <method-name>METHOD</method-name>
        <method-params>
                <method-param>PARAM-1</method-param>
                <method-param>PARAM-2</method-param>
                              ...
                <method-param>PARAM-n</method-param>
        </method-params>
      <method>

   This style is used to refer to a single method within a set of
   methods (home and remote interfaces) with an overloaded name for all
   deployments in a container-system. PARAM-1 through PARAM-n are the
   fully-qualified Java types of the method's input parameters (if the method
   has no input arguments, the method-params element contains no method-param
   elements). Arrays are specified by the array element's type, followed by
   one or more pair of square brackets (e.g. int[][]).


   The optainal ejb-deployment-id element can be used to narrow the scope of the
   declaration to one specific deployment within the container-system.
   If an ejb-deployment-id is not specified, the declaration applies to all
   matching bean methods (home and remote interface) in all deployments.

   The optional method-intf element can be used when it becomes
   necessary to differentiate between a method defined in the home
   interface and a method with the same name and signature that is
   defined in the remote interface.
-->
<!ELEMENT method (description?, ejb-deployment-id?, method-intf?, method-name, method-
params?)>

<!ELEMENT method-intf (#PCDATA)>
<!ELEMENT method-name (#PCDATA)>
<!ELEMENT method-param (#PCDATA)>
<!ELEMENT method-params (method-param*)>
<!ELEMENT method-permission (description?, role-name+, method+)>

<!--
The method-transaction element specifies how the container must
manage transaction scopes for the enterprise bean's method invocations.
The element consists of an optional description, a list of method
elements, and a transaction attribute.The transaction attribute is to
be applied to all the specified methods.

Used in: container-system
Maps to: container-transaction ejb-jar_1_1
-->
<!ELEMENT method-transaction (description?, method+, trans-attribute)>
```

```
<!ELEMENT openejb (container-system, facilities)>
<!ELEMENT physical-role-name (#PCDATA)>
<!ELEMENT primary-key (#PCDATA)>
<!ELEMENT properties (property+)>

<!ELEMENT property (property-name, property-value)>

<!ELEMENT property-name (#PCDATA)>
<!ELEMENT property-value (#PCDATA)>
<!ELEMENT proxy-factory (#PCDATA)>
<!ELEMENT role-mapping (logical-role-name+, physical-role-name+)>
<!ELEMENT role-name (#PCDATA)>

<!--
The role-link element is used to link a security role reference to a
defined security role. The role-link element must contain the name of
one of the security roles defined in the security-role elements.
Used in: security-role-ref
-->
<!ELEMENT role-link (#PCDATA)>


<!ELEMENT remote (#PCDATA)>
<!--
The res-auth element specifies whether the enterprise bean code signs
on programmatically to the resource manager, or whether the Container
will sign on to the resource manager on behalf of the bean. In the
latter case, the Container uses information that is supplied by the
Deployer.
The value of this element must be one of the two following:
<res-auth>Application</res-auth>
<res-auth>Container</res-auth>
Mapps to: res-auth ejb-jar_1_1
-->
<!ELEMENT res-auth (#PCDATA)>
<!--
The res-id maps to a connector-id in the corresponding connectors section. For example a
JDBC DataSource resource-ref would be supported by a connector and its res-id would map
to the connector-id in connector declaration.
Used in: resource-ref
-->
<!ELEMENT res-id (#PCDATA)>
<!--
The res-ref-name element specifies the name of a resource manager con-nection
factory reference.
Used in: resource-ref
Mapps to: res-ref-name ejb-jar_1_1
-->
<!ELEMENT res-ref-name (#PCDATA)>
<!--
The res-type element specifies the type of the data source. The type
is specified by the Java interface (or class) expected to be imple-mented
by the data source.
Used in: resource-ref
Mapps to: res-type ejb-jar_1_1
-->
<!ELEMENT res-type (#PCDATA)>
<!--
The resource-ref element contains a declaration of enterprise bean's
reference to an external resource. It consists of an optional description,
the resource factory reference name, the indication of the
resource manager connection factory type expected by the enterprise
bean code, the type of authentication (bean or container), and either a
```

```
res-id which maps to the corresponding services res-id, or a set of properties
Used in: jndi-enc
Example:
<resource-ref>
    <res-ref-name>comp/env/jdbc/Employee</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <properties>
        <property>
            <property-name>url</property-name>
            <property-value>jdbc:odbc:orders</property-value>
        </property>
        <property>
            <property-name>username</property-name>
            <property-value>Admin</property-value>
        </property>
        <property>
            <property-name>password</property-name>
            <property-value></property-value>
        </property>
    </properties>
</resource-ref>
Mapps to: resource-ref ejb-jar_1_1
-->
<!ELEMENT resource-ref (description?, res-ref-name, res-type, res-auth, (res-id |
properties | connector-id))>

<!ELEMENT resource (description?, res-id, properties)>

<!ELEMENT security-role (description?, role-name)>
<!--
The security-role-ref element contains the declaration of a security
role reference in the enterprise bean's code. The declaration con-sists
of an optional description, the security role name used in the
code, and an optional link to a defined security role.
The value of the role-name element must be the String used as the
parameter to the EJBContext.isCallerInRole(String roleName) method.
The value of the role-link element must be the name of one of the
security roles defined in the security-role elements.
Used in: entity and session
-->
<!ELEMENT security-role-ref (description?, role-name, role-link)>

<!ELEMENT security-service (description?, display-name?, service-name, factory-class,
codebase, properties?, role-mapping+)>
<!ELEMENT security-service-name (#PCDATA)>
<!ELEMENT services (security-service, transaction-service)>
<!ELEMENT service-name (#PCDATA)>
<!ELEMENT small-icon (#PCDATA)>
<!ELEMENT stateful-bean (description?, display-name?, small-icon?,large-icon?, ejb-
deployment-id, home, remote, ejb-class, transaction-type, jndi-enc?, security-role-ref*)>
<!ELEMENT stateless-bean (description?, display-name?, small-icon?,large-icon?, ejb-
deployment-id, home, remote, ejb-class, transaction-type, jndi-enc?, security-role-ref*)>
<!ELEMENT stateful-session-container (description?, display-name?, container-name,
properties?, stateful-bean+)>
<!ELEMENT stateless-session-container (description?, display-name?, container-name,
properties?, stateless-bean+)>
<!ELEMENT transaction-service (description?, display-name?, service-name, factory-class,
codebase, properties?)>
<!ELEMENT transaction-service-name (#PCDATA)>
<!ELEMENT transaction-type (#PCDATA)>

<!--
```

```
The trans-attribute element specifies how the container must manage the
transaction boundaries when delegating a method invocation to an
enterprise bean's business method.

The value of trans-attribute must be one of the following:
        <trans-attribute>NotSupported</trans-attribute>
        <trans-attribute>Supports</trans-attribute>
        <trans-attribute>Required</trans-attribute>
        <trans-attribute>RequiresNew</trans-attribute>
        <trans-attribute>Mandatory</trans-attribute>
        <trans-attribute>Never</trans-attribute>
        <trans-attribute>Bean</trans-attribute> (indicates bean managed transaction.
Stession beans only)
Used in: method-transaction
Maps to: trans-attribute ejb-jar_1_1
-->

end of file: openejb_config.dtd
```

In the foregoing, the method and apparatus of the present invention is described with reference to specific exemplary embodiments thereof. It will, however, be evident that various modifications and changes may be made thereto without departing from the broader spirit and scope of the present invention. In particular, the separate blocks of the various block diagrams represent functional blocks of methods or apparatuses and are not necessarily indicative of physical or logical separations or of an order of operation inherent in the spirit and scope of the present invention. For example, the various blocks of some figures or illustrations may be integrated into components, or may be subdivided into components. Moreover, the various blocks of other figures or illustrations represent portions of a method which, in some embodiments, may be reordered or may be organized in a parallel or a linear or step-wise fashion. The present specification and figures or illustrations are accordingly to be regarded as illustrative rather than restrictive.

CLAIMS

What is claimed is that which has been described in the foregoing and equivalents thereof.